Summer 8-31-2011

Security systems based on Gaussian integers : Analysis of basic operations and time complexity of secret transformations

Aleksey Koval
New Jersey Institute of Technology

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

SECURITY SYSTEMS BASED ON GAUSSIAN INTEGERS: ANALYSIS OF BASIC OPERATIONS AND TIME COMPLEXITY OF SECRET TRANSFORMATIONS

by
Aleksey Koval

Many security algorithms currently in use rely heavily on integer arithmetic modulo prime numbers. Gaussian integers can be used with most security algorithms that are formulated for real integers. The aim of this work is to study the benefits of common security protocols with Gaussian integers. Although the main contribution of this work is to analyze and improve the application of Gaussian integers for various public key (PK) algorithms, Gaussian integers were studied in the context of image watermarking as well.

The significant benefits of the application of Gaussian integers become apparent when they are used with Discrete Logarithm Problem (DLP) based PK algorithms. In order to quantify the complexity of the Gaussian integer DLP, it is reduced to two other well known problems: DLP for Lucas sequences and the real integer DLP. Additionally, a novel exponentiation algorithm for Gaussian integers, called Lucas sequence Exponentiation of Gaussian integers (LSEG) is introduced and its performance assessed, both analytically and experimentally. The LSEG achieves about 35% theoretical improvement in CPU time over real integer exponentiation. Under an implementation with the GMP 5.0.1 library, it outperformed the GMP's "mpz_powm" function (the particularly efficient modular exponentiation function that comes with the GMP library) by 40% for bit sizes 1000-4000, because of low overhead associated with LSEG. Further improvements to real execution time can be easily achieved on multiprocessor or

multicore platforms. In fact, over 50% improvement is achieved with a parallelized implementation of LSEG. All the mentioned improvements do not require any special hardware or software and are easy to implement. Furthermore, an efficient way for finding generators for DLP based PK algorithms with Gaussian integers is presented.

In addition to DLP based PK algorithms, applications of Gaussian integers for factoring-based PK cryptosystems are considered. Unfortunately, the advantages of Gaussian integers for these algorithms are not as clear because the extended order of Gaussian integers does not directly come into play. Nevertheless, this dissertation describes the Extended Square Root algorithm for Gaussian integers used to extend the Rabin Cryptography algorithm into the field of Gaussian integers. The extended Rabin Cryptography algorithm with Gaussian integers allows using fewer preset bits that are required by the algorithm to guard against various attacks. Additionally, the extension of RSA into the domain of Gaussian integers is analyzed. The extended RSA algorithm could add security only if breaking the original RSA is not as hard as factoring. Even in this case, it is not clear whether the extended algorithm would increase security.

Finally, the randomness property of the Gaussian integer exponentiation is utilized to derive a novel algorithm to rearrange the image pixels to be used for image watermarking. The new algorithm is more efficient than the one currently used and it provides a degree of cryptoimmunity. The proposed method can be used to enhance most picture watermarking algorithms.

SECURITY SYSTEMS BASED ON GAUSSIAN INTEGERS: ANALYSIS OF BASIC OPERATIONS AND TIME COMPLEXITY OF SECRET TRANSFORMATIONS

by
Aleksey Koval

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

August 2011

.

APPROVAL PAGE

SECURITY SYSTEMS BASED ON GAUSSIAN INTEGERS: ANALYSIS OF BASIC OPERATIONS AND TIME COMPLEXITY OF SECRET TRANSFORMATIONS

Aleksey Koval

Dr. Boris Verkhovsky, Dissertation Advisor — Date
Professor of Computer Science, NJIT

Dr. Frank Shih, Committee Member — Date
Professor of Computer Science, NJIT

Dr. Cristian Borcea, Committee Member — Date
Associate Professor of Computer Science, NJIT

Dr. James Geller, Committee Member — Date
Professor of Computer Science, NJIT

Dr. Joon Sung, Committee Member — Date
Technical Manager, IBM, AT&T Labs,Middletown, NJ

BIOGRAPHICAL SKETCH

Author: Aleksey Koval

Degree: Doctor of Philosophy

Date: August 2011

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2011
- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2009
- Master of Science in Applied Statistics,
  Rutgers University, New Brunswick, NJ, 1999
- Bachelor of Science in Computer Science,
  Kean University, Union, NJ, 1997
- Bachelor of Science in Mathematics,
  Kean University, Union, NJ, 1997

Major: Computer Science

Presentations and Publications:

A. Koval, F. Y. Shih, and B. S. Verkhovsky, "A Pseudo-Random Pixel Rearrangement Algorithm Based on Gaussian Integers for Image Watermarking," *Journal of Information Hiding and Multimedia Signal Processing,* vol. 2, no. 1, pp. 60-70, 2010.

A. Koval, "On Lucas Sequences Computation," *Int'l J. of Communications, Network and System Sciences* vol. 2, no. 12, pp. 943-944 2010.

A. Koval, and B. S. Verkhovsky, “On Discrete Logarithm Problem for Gaussian Integers,” in International Conference on Information Security and Privacy (ISP-09), Orlando, Florida, USA, 2009, pp. 79-84.

A. Koval, and B. Verkhovsky, “Analysis of RSA over Gaussian Integers Algorithm,” in Fifth International Conference on Information Technology: New Generations (ITNG 2008), Las Vegas, Nevada, USA, 2008, pp. 101-105.

B. Verkhovsky, and A. Koval, “Cryptosystem Based on Extraction of Square Roots of Complex Integers,” in Fifth International Conference on Information Technology: New Generations (ITNG 2008), Las Vegas, Nevada, USA, 2008, pp. 1190-1191.

This dissertation is dedicated to the memory of my father

Dr. Yevgeniy Aleksandrovich Koval.

ACKNOWLEDGMENT

I wish to thank my advisor, Dr. Boris Verkhovsky for all his guidance, dedication, patience and support. His sincere curiosity in the unexplored fields of cryptography inspired me. His dedication and professionalism helped me overcome all of the obstacles.

In addition, I would like to thank the members of my committee Dr. Frank Shih, Dr. Cristian Borcea, Dr. James Geller, and Dr. Joon Sung. Dr. James Geller spent a lot of his time helping me improve this dissertation. I really appreciate the insightful observations of Dr. Christian Borcea that directed my work in the right direction. Also, I appreciate Dr. Frank Shih's teaching and guidance, especially on image related topics. Dr. Joon Sung provided great advice, criticism and encouragement.

I would like to thank Dr. Dimitri Kanevsky (IBM T.J Watson Research Center) for his support and guidance.

TABLE OF CONTENTS

Chapter		Page
1 INTRODUCTION		1
1.1	Problem Statement…….	4
1.2	Survey of References…….	6
1.3	Overview of Gaussian Integers, Notation and Definitions……	14
1.4	Dissertation Structure……	22
2 DISCRETE LOGARITHM CRYPTOGRAPHY WITH GAUSSIAN INTEGERS……...		24
2.1	Gaussian Primes P: $\|P\|$ is a non-Blum Prime……..…	24
2.2	Common Cryptography Algorithms Based on Discrete Logarithm…….…..	32
2.3	Properties of Gaussian Integer Exponentiation……..….	37
2.4	Discrete Logarithm Complexity for Gaussian Integers……..	42
2.5	Reducing Gaussian Integer DLP to Lucas Sequences DLP……..	47
2.6	Multiplication of Gaussian Integers vs. Real Integer Multiplication…….	54
2.7	Computation of Lucas Sequences……..	70
2.8	Exponentiation of Gaussian Integers…….	72
2.9	Experimental Results……..….	82
2.10	Algorithms for Finding Gaussian Generators……..…	89
2.11	Chapter Summary……...	95
3 EXTENSION OF RABIN CRYPTOSYSTEM INTO THE FIELD OF GAUSSIAN INTEGERS…….		97
3.1	Restriction of Gaussian Integer Domain……	97

TABLE OF CONTENTS
(Continued)

Chapter **Page**

3.2 Rabin Cryptosystem……..... 97

3.3 Square Roots Modulo $n=pq$…….. 99

3.4 Extended Square Root Algorithm mod p…….. 101

3.5 Extended Square Root Algorithm mod $n=pq$……... 109

3.6 Extended Rabin Cryptosystem……..... 111

3.7 Security of the Extended Rabin Cryptosystem……. 112

3.8 Chapter Summary……. 114

4 ANALYSIS OF RSA ALGORITHM OVER GAUSSIAN INTEGERS…… 115

4.1 Description of RSA Algorithm over the Field of Gaussian Integers…… 115

4.2 Cryptanalysis of RSA Algorithm over the Field of Gaussian Integers……. 116

4.3 Chapter Summary…….. 127

5 A PSEUDO-RANDOM PIXEL REARRANGEMENT ALGORITHM BASED ON GAUSSIAN INTEGERS FOR IMAGE WATERMARKING……..... 129

5.1 Algorithm Introduction……. 129

5.2 Proposed Pixel Rearrangement Algorithm…… 131

5.3 Cryptoimmunity of the Rearrangement Algorithm……... 135

5.4 Comparison to Arnold's Cat Map Chaos Transformation…… 138

5.5 Example in Image Watermarking……. 142

5.6 Chapter Summary……. 144

6 CONCLUSION……... 145

REFERENCES…… 149

LIST OF TABLES

Table		Page
2.1	Discrete Power Table MOD P=3+2i, $\|P\|$=13, $\sqrt{-1} \bmod 13 = 5$.....	31
2.2	Repeating Norm Example for Prime p=7……..……..……..	37
2.3	Repeating Norm Example for Prime p=11…….	38
2.4	Gaussian Integer Exponentiation and Lucas Sequences…….	52
2.5	Summarized Estimates of the Multiplication Running Time Ratio Based on the Formula (2.116)……..	68
2.6	Summarized Estimates of the Square Running Time Ratio Based on the Formula (2.117)……..	68
2.7	T_{LSEG} / T_{SWG} Ratio for Various β and Window Sizes…….	79
2.8	T_{LSEG^*} / T_{SWG} Ratio for Various β and Window Sizes……	81

LIST OF FIGURES

Figure		Page
2.1	The ratio of the running time of multiplication of two numbers of the equal size vs. the running time of square of a number of the same size. The graph represents a typical performance of GMP 5.0.1 library on various platforms.....	59
2.2	The distribution of optimal multiplication thresholds among various platforms for GMP 5.0.1.....	61
2.3	The distribution of optimal square thresholds among different platforms and counts for GMP 5.0.1.....	62
2.4	Running time of mod operation versus multiplication using GMP 5.0.1 library on AMD Opteron Model 2218 @2.6 GHz Dual core, 8GB of RAM, RHEL Linux 4.2 kernel 2.6.9 (64 bit).....	66
2.5	Running time of mod operation divided by the running time of multiplication using GMP 5.0.1 library on AMD Opteron Model 2218 @2.6 GHz Dual core, 8GB of RAM, RHEL Linux 4.2 kernel 2.6.9 (64 bit).....	66
2.6	The CPU time of SWR,SWG, LSEG and LSEG* for various bit sizes…….....	85
2.7	The ratio of the running time of SWG algorithm over SWR.....	86
2.8	The ratio of the CPU time of Algorithm 2.8.1 (LSEG) over SWG.....	87
2.9	The ratio of the running time of Algorithm 2.8.1 algorithm over SWR.....	87
2.10	The real running time of SWR, SWG, LSEG and LSEG* for various bit sizes.....	88
2.11	Ratio of real running time of LSEG* over SWG……	89
5.1	Image rearranged by Algorithm 5.2.1 and Arnold's Cat map side-by-side. A is the original image, B is the rearranged image by Algorithm 5.2.1, and C1-C7 are the steps of Arnold's Cat map rearrangement.....	141

LIST OF FIGURES
(Continued)

Figure **Page**

5.2 (a) The original Cameraman image, (b) the two most significant bits of Lena as the watermark, (c) the rearranged image of Cameraman using Algorithm 5.2.1, (d) the watermarked image of the rearranged image using LSB substitution, (e) the rearranged back of the preceding watermarked image using Algorithm 5.2.2, (f) the extracted two bits of LSB (g) the rearranged back of the preceding extracted image using Algorithm 5.2.2..... 143

CHAPTER 1

INTRODUCTION

The history of cryptography dates back thousands of years. Over most of this time, it has been a history of symmetric cryptography. It appeared obvious that the only way for several parties to communicate securely is to share a secret method or a key. It seemed that there is no other way because the recipient must have an advantage over eavesdropper. Key exchange is the weakest link of symmetric cryptography. The challenge of exchanging secret keys securely is magnified when there are many parties that need to communicate.

The revolution in cryptography happened in 1970s when Public Key or asymmetric cryptography was introduced. In 1976, Diffie and Hellman published a revolutionary paper titled "New Directions in Cryptography" [26], where they introduced the concepts of Public Key or asymmetric cryptography. In addition, they introduced the method of exchanging keys known as Diffie-Hellman Key Exchange protocol. The Diffie-Hellman Key Exchange protocol relies on the difficulty of the discrete logarithm problem. Similar techniques were invented earlier by James H. Ellis, Clifford Cocks, and Malcolm Williamson at GCHQ but were kept secret until the late 1990s. After this, many new Public Key algorithm and techniques were introduced. Most notable of these are RSA, Rabin, ElGamal and Elliptic Curve Cryptography (ECC).

In 1977, the RSA algorithm was invented by Rivest, Shamir and Adleman at MIT. It relies on the difficulty of factoring large numbers, which are products of two

large primes. RSA was a great success and currently is the most commonly used Public Key Encryption algorithm.

In 1979, M. O. Rabin introduced a Rabin Cryptosystem, which, as RSA, is based on the difficulty of factoring large numbers. Rabin Cryptosystem has some notable advantages over RSA; however, it is not as widely used as RSA.

In 1984, Taher ElGamal introduced the ElGamal algorithm. As the Diffie-Hellman Key Exchange protocol, it is using the difficulty of the discrete logarithm problem. As RSA, ElGamal is currently widely used.

In 1985, Neal Koblitz and Victor S. Miller introduced Elliptic Curve Cryptography (ECC). It uses a special algebraic structure called elliptic curves over finite group. ECC is very promising technique because the discrete logarithm problem over elliptic curves is more difficult than the same problem over integers. This allows for smaller keys which, in turn, increase the efficiency. ECC has been recommended by the NSA and seem to have a very bright future.

Despite apparent advantages of Public Key cryptography, it is not about to replace symmetric cryptography. There are many reasons to use symmetric cryptography. The most important one is that all known Public Key algorithms are not nearly as efficient as symmetric cryptography algorithms. For instance, asymmetric algorithms may work well to encrypt emails because a delay of fraction of a second for email is not noticeable. However, for real time delay sensitive applications like Voice over IP (VOIP) or Virtual Private Networks (VPN) this kind of delay is unacceptable. The practical solution for this is to use a Private Key algorithm to distribute a symmetric key and use the symmetric key to encrypt and decrypt the messages. For example, the contemporary VPN protocols use

Diffie-Hellman Key Exchange protocol (asymmetric) to exchange Triple DES (symmetric protocol) keys.

Efficiency of Public Key algorithms is directly tied to the size of the key. As computing power grows, the keys have to grow also. For example, 512-bit keys for RSA were considered sufficiently secure. At present, even 1024 bit keys are sometimes considered potentially weak. Most companies and individuals use 2048 bit keys for RSA now.

One of the directions of contemporary cryptography research is extending tried-and-true Public Key Cryptography algorithms such as RSA, ElGamal and Rabin into well-studied cyclical groups. The aim is to improve the security of the algorithms by introducing more complexity. Improved security would allow for use of smaller keys, in turn, improving efficiency. One difficulty is that with increased complexity overhead is introduced that may undermine any efficiency improvements. Another difficulty is that as the algorithms become more complex it becomes harder to assess their security.

In this dissertation, the use of Gaussian integers as the underlying field for RSA, ElGamal and Rabin algorithms is studied. The extension of the Rabin algorithm into the field of Gaussian integers is introduced.

Gaussian integers are complex numbers with integers as both real and imaginary part. Carl Friedrich Gauss introduced the ring of Gaussian integers in 1829 – 1831. He formulated many properties of Gaussian integers like properties of factorization and the concept of Gaussian Prime. Gauss used them as a tool to prove some theoretical results. The properties of Gaussian integers and Gaussian Primes are well known and formulated so they are going to be used as facts.

1.1 Problem Statement

Many security algorithms currently in use rely heavily on integer arithmetic modulo prime numbers. Gaussian integers can be used with most security algorithms that are formulated for real integers. The aim of this work is to study the benefits of common security protocols with Gaussian integers. Although the main contribution of this work is to analyze and improve the application of Gaussian integers for various public key (PK) algorithms, Gaussian integers were studied in the context of image watermarking as well.

Among the most widely used PK algorithms are RSA, Diffie-Hellman key exchange, ElGamal, and Rabin [58] PK algorithms. Unfortunately, in order to provide a required degree of cryptoimmunity, the keys must be very large. Large keys mean lower speed of encryption/decryption/authentication. One of the ways to increase speed is to consider more complicated fields with larger cyclic groups, e.g., Gaussian integers. Most mainstream PKC algorithms fall into two categories: Discrete Logarithm problem (DLP) based (e.g., ElGamal or Diffie-Hellman key exchange) and integer factoring based (RSA or Rabin). Gaussian integers can be successfully used with all the PK algorithms that are formulated for real integers and this work explores the application of Gaussian integers for both types of PK algorithms.

The Gaussian integer modulo prime cyclic group order is much larger then the real integer modulo prime order for the same prime. However, larger order does not guarantee increased security nor does it mean that the extended PK algorithms would be more efficient. The security depends on the complexity of the underlying DLP. Unfortunately, assessing complexity of such DLP is usually very hard. One way to do it is to reduce the Gaussian integer DLP to another well known problem: DLP for Lucas

sequences, which is about twice as hard as the real integer DLP for the same prime. This reduction is described in Chapter 2. Another challenge was to perform the exponentiation of Gaussian integers faster than the exponentiation of real integers. This goal was achieved with a novel exponentiation algorithm for Gaussian integers, which called Lucas sequence Exponentiation of Gaussian integers (LSEG). The performance of LSEG is assesed both analytically and experimentally. The LSEG achieves about 35% theoretical improvement in CPU time over real integer exponentiation. Under an implementation with the GMP 5.0.1 library it outperformed the GMP's "mpz_powm" function (the particularly efficient modular exponentiation function that comes with the GMP library) by 40% for bit sizes 1000-4000, because of low overhead associated with LSEG. Further improvements to real execution time can be easily achived on multiprocessor or multicore platforms with parallelizing certain steps in LSEG. All the mentioned improvements do not require any special hardware or software and are easy to implement. Additionally, an efficient way for finding generators is proposed. It would be useful for real-world implementations of DLP based PK algorithms with Gaussian integers.

In addition to DLP based PK algorithms, the applications of Gaussian integers for factoring-based PK cryptosystems are considered. Unfortunately, the advantages of Gaussian integers for these algorithms are not as clear, because the extended order of Gaussian integers does not directly come into play. Nevertheless, the Extended Square Root algorithm for Gaussian integers is derived and its validity is proven. Using this algorithm, Rabin Cryptography algorithm was extended into the field of Gaussian integers. The resulting Extended Rabin Cryptography algorithm allows using fewer

preset bits that are required by the algorithm to guard against various attacks. Additionally, the extension of RSA into the domain of Gaussian integers is analyzed in-depth. The analysis, published in [49], yielded several interesting results, e.g., that a certain type of Gaussian primes does not offer any advantages over real primes.

Finally, the randomness property of the Gaussian integer exponentiation is utilized to derive a novel algorithm to rearrange the image pixels to be used for image watermarking. Currently many image watermarking techniques use Arnold's cat map to rearrange the image pixels as a part of the watermarking algorithm. In the rearrangement step, Arnold's cat map can be replaced with the new algorithm based on Gaussian integers, which has the advantages of increased speed and security. Moreover, the new algorithm can provide a degree of cryptoimmunity to image watermarking. The proposed method can be used with most picture watermarking algorithms to enhance them.

The techniques and theoretical framework developed and presented in this dissertation offer some interesting avenues for further research. Potential uses include new cryptography algorithms, primality testing, steganography and cryptanalysis of the existing algorithms.

1.2 Survey of References

In 1979, M. O. Rabin in his paper "Digitalized Signatures and Public Key Functions as Intractable as Factorization", [58], introduced a new cryptosystem, later called the Rabin Cryptosystem. The Rabin Cryptosystem, as the RSA, is based on the difficulty of factoring large numbers. Rabin Cryptosystem has some notable advantages over the

RSA, mainly faster encryption. The encryption with Rabin is very simple. If m is a message and $n=pq$ is a product of two large primes, then the ciphertext c is $c=m^2 \bmod n$. To decrypt the message, the reverse operation is needed, namely, the receiver has to take a square root of c mod n. Rabin showed that the square root mod n operation is equivalent to factoring of n. This means that the code can only be broken if the adversary can factor n. Thus the Rabin Cryptosystem is proven as secure as factorization.

As other public key cryptosystems, the Rabin Cryptosystem can be used to digitally sign documents. The method for signing documents using public key cryptosystems was first described in the seminal paper by R. L. Rivest. A. Shamir, and L. Adleman: "A Method for Obtaining Digital Signatures and Public Key Cryptosystems", [59], where the authors introduced the concept of digital signatures.

In 1985, W. Alexi, B. Chor, O. Goldreich and C. P. Schnorr published "RSA/Rabin Functions: Certain Parts are As Hard As the Whole", [4], where they prove that, if one is able to predict the least significant bit of the number $m^2 \bmod n$ (Rabin) or the $m^e \bmod n$ (RSA) with a probability greater than ½, then it is possible to invert the function. This result is important for algorithms that use Rabin or RSA for random number generators.

Another notable paper on the subject of Rabin algorithm signatures security is "Proving Tight Security for Standard Rabin-Williams Signatures", [13], by Daniel J. Bernstein. In this paper, the author proves that any generic attack on standard Rabin signatures could be converted into the factorization algorithm, thus proving the security of Rabin signatures.

In 2001, A. N. El-Kassar, M. Rizk, N. Mirza, and Y.A. Awad, in a paper titled "ElGamal Public-Key Cryptosystem in the Domain of Gaussian Integers" [27] introduced an extension of the ElGamal algorithm into the field of Gaussian integers. The extension deals with Gaussian integers modulo real Gaussian Primes (primes $p : p \bmod 4 = 3$). The proposed cryptosystem is, presumably, more secure because the order of a Gaussian Prime generator is p^2-1 as opposed to p for real integers. This is potentially a huge advantage because this allows for the use of smaller primes, which dramatically improves the efficiency.

In 2002, H. Elkamchouchi, K. Elshenawy and H. Shaban introduced the extension of the RSA algorithm to the field of Gaussian integers in their paper "Extended RSA Cryptosystem and Digital Signature Schemes in the Domain of Gaussian Integers" [30]. As opposed to the ElGamal extension, the domain of Gaussian Primes is not restricted. Consequently, the strength of this algorithm is based on Gaussian integer factoring as opposed to real integer factoring. The security of the proposed cryptosystem was not proven in this paper.

In 2004, A. N. El-Kassar, R. A. Haraty and Y.A. Awad in their paper "Modified RSA in the Domains of Gaussian Integers and Polynomials Over Finite Fields" [28] formulated the extension of RSA into the domain of Gaussian integer modulo real primes similar to the domain in [27]. This paper describes a special case of the extended RSA algorithm described in [30].

In 2004, Ramzi A. Haraty, A. N. El-Kassar and Hadi Otrok in their paper "A Comparative Study of RSA based Cryptographic Algorithms" [35] tested the reliability and security of several RSA extensions described in [28]. The authors found that all

algorithms tested to be reliable and probably secure. The running time of Gaussian RSA was similar to the original RSA. This paper does not prove the security of Gaussian integer RSA.

In 2004, Ramzi A. Haraty, Hadi Otrok and A. N. El-Kassar in their paper "A Comparative Study of ElGamal Based Cryptographic Algorithms" [36] tested the reliability and security of several extensions of the ElGamal algorithm. Among the algorithms tested, was an extension of ElGamal into the field of Gaussian integers described in [27]. To test the security the Baby-step Giant-step algorithm was used. The authors found that the ElGamal algorithm with Gaussian integers was probably stronger than the original, because the discrete logarithm took for Gaussian integers took twice as long to compute. By no means is this a proof that it is strong, however, it is an indication that it could be stronger than the original.

The paper by Ramzi A. Haraty, Hadi Otrok and A. N. El-Kassar "Attacking ElGamal Based Cryptographic Algorithms Using Pollard's Rho Algorithm" [38] is very similar to [36]. Here, to test the security the authors enhanced the Pollard's Rho algorithm to work with Gaussian integers (the original Pollard's Rho algorithm works with real integers). All the analysis and results are essentially the same as in [36].

In 2005, Boris S. Verkhovsky and A. Mutovic in their paper "Primality Testing Algorithm Using Pythagorean Integers" [66] introduced a novel use for Gaussian integers, namely, primality testing. The algorithms presented improve the performance of the Fermat’s original primality test. They are able to detect Carmichael numbers (undetectable with the original Fermat’s test) with high probability. The primality test introduced in [67] uses quaternions to further improve the probability of detecting

Carmichael numbers. The theory and techniques that will be presented in this dissertation, together with other ideas presented by Dr. Boris S. Verkhovsky, may allow an improvement of the test introduced in [66]. The primality testing with Gaussian integers and their variants will not be in the scope of this dissertation, but illustrates the practical value of the topic to be explored.

The paper by Ramzi A. Haraty, A. N. El-Kassar and B. Shibaro "A Comparative Study of RSA Based Digital Signature Algorithms" [37] is very similar to [35]. As opposed to encryption and decryption in [35], this paper deals with extended RSA digital signature schemes. For the most part, it is a report on experiments ran by the authors.

The paper by Peter Smith “LUC Public Key Encryption: a Secure Alternative to RSA” [62], published in 1993, describes the first cryptosystem that is based on Lucas sequences, called LUC. LUC uses calculation of Lucas functions as an alternative to real integer exponentiation. The paper claims that “while Lucas functions are somewhat more complex mathematically than exponentiation, they produce superior ciphers. “

Another paper by Peter Smith “Cryptography Without Exponentiation” [63], published in 1994, introduced three more algorithms based on Lucas sequences: a Lucas-function ElGamal PK encryption, a Lucas-function ElGamal digital signature, and a key exchange algorithm called LUCDIF (essentially, LUCDIF is the Diffie-Hellman key exchange protocol over Lucas sequences). All three algorithms are based on the difficulty of the Discrete Logarithm problem for Lucas functions. The author claims that the proposed cryptosystems are stronger, because they are not based on exponentiation and, therefore, the subexponential-time algorithms currently known cannot be used against them.

The paper by Chi-Sung Laih, Fu-Kuan Tu and Wen-Chung Tai "On the Security of the Lucas Function" [51], published in 1995, discusses the security of Discrete Logarithms for Lucas sequences. The authors raised doubts about the hypothesis that the security of the Lucas function is cryptographically stronger than or at least as strong as the security of the exponentiation function. They also show that the security of the Lucas function is polynomial-time equivalent to the generalized discrete logarithm problems.

The paper by Arjen K. Lenstra, Daniel Bleichenbacher and Wieb Bosma "Some Remarks on Lucas-Based Cryptosystems" [52], published in 1995, discusses the security of all Lucas sequence-based cryptosystems. For LUC it describes a chosen ciphertext attack, as a result proving that LUC is not stronger than RSA. Additionally, a subexponential attack on Discrete Logarithm for Lucas sequences is described.

The computation of Lucas sequences is a very important subject of this dissertation. The first significant paper was published on the subject in 1995 by S.M. Yen and C.S. Laih "Fast Algorithms for LUC Digital Signature Computation" [74]. The paper describes two efficient algorithms to compute Lucas sequences for LUC cryptographic algorithms. The two algorithms are analogous to square-multiply algorithms for real integers. Logical extensions of the algorithms published in [74] is represented by the work by C.S. Laih and S.Y. Chiou "An Efficient Algorithm for Computing the Luc Chain" first published in 1995 ([18]) and later published again in [19]. It describes an improvement to [74] that is achieved by using addition chains for LUC exponentiation. Another significant paper that introduces improvements of [74] by using addition chains is the paper by C.T. Wang, C.C. Chang, and C.H. Lin "A Method for Computing Lucas Sequences" published in 1999 [68]. Incidentally, quite a few papers have been published

on this subject recently, namely [5-9, 56, 57], that do not describe any improvements to [68].

The LUC cryptosystem is based on one of the two Lucas sequences, namely *V*. The computation of both Lucas "sister" sequences is of particular interest. Such an algorithm was published in 1996 in the paper "Efficient Computation of Full Lucas Sequences" by M. Joye and J. J. Quisquater. The improved algorithm was published in [47]. It is utilized as an alternative to the Gaussian integer exponentiation.

For this discussion, the complexity of the multiplication operation is very important. Depending on the integer size, different multiplication methods are appropriate. For small bit sizes the naïve multiplication method [44] with complexity of $\Theta(n^2)$ is most efficient. For larger bit sizes the Karatsuba-Ofman [43] multiplication algorithm is universally used. As bit sizes increase, multiple levels of the *k*-way Toom-Cook multiplication ([21],[44]) could be applied. For extremely large bit sizes, algorithms based on Fast Fourier transforms (FFT) such as the Schönhage–Strassen algorithm ([60]) and Fürer's algorithm ([31]) become practical. Since FFT algorithms are used for very large bit sizes, the FFT algorithms will not be considered in the subsequent discussion.

Another important topic is the time complexity of modular reduction. The performance relative to multiplication is of particular interest. The "mod" division operation is much slower than multiplication. For small to moderate integer sizes, the divide-and-conquer algorithm [16] is commonly used for modular division. However, for efficient modular exponentiation algorithms the costly *mod* operation is replaced with the Montgomery reduction or *REDC*() operation ([55]), because it is much more efficient

([14]). There are quite a few implementation variations for Montgomery reduction analyzed in [46] and [17]. Moreover, there are many papers published with marginal improvements to the Montgomery reduction method, most of them through low level implementations and specialized hardware (e.g., [1, 11, 20, 23, 29, 39, 71]). The performance of the reduction algorithms (either modular division or Montgomery *REDC*) relative to multiplication is of interest in this discussion. In particular, the range from one to four multiplications in which all of the contemporary reduction implementations fall is considered.

Steganography is a process of hiding information in a medium in such a manner that no one except the anticipated recipient knows of its existence ([61]). A notable application of steganography is watermarking of digital images, which is a useful tool for identifying the source, creator, owner, distributor, or authorized consumer of a document or an image. A way to apply Gaussian integers for image watermarking is described in this dissertation. There are many innovative watermarking algorithms and many more get published every day (such as recently published [3, 41, 53, 70]). In many image watermarking algorithms, for example in [24, 69, 72, 73], it is required to rearrange the pixels as a part of the watermarking process. An algorithm that uses Gaussian integers for the rearrangement step is presented in [48].

Gaussian integers and Gaussian primes have a long history and have been studied as a mathematical subject. However, only recently they have been used to extend popular Public Key cryptography algorithms. The published papers directly related to the proposed topic are [27, 28, 30, 35, 36, 38]. The two most common Public Key cryptography algorithms RSA and ElGamal have been extended into the field of

Gaussian integers ([30] and [27]). An extension of another classic cryptography algorithm, Rabin, is presented in this dissertation. Most of the papers published state that the extended cryptosystems have advantages over the corresponding real integer algorithms. However, none of them prove or carefully analyze these statements. This dissertation would close many of the gaps in the subject.

1.3 Overview of Gaussian Integers, Notation and Definitions

Gaussian integer is a complex number $a+bi$ where both a and b are integers:

$$Z[i] = \{a + bi : a, b \in \mathbb{Z}\} \tag{1.1}$$

Gaussian integers, with ordinary addition and multiplication of complex numbers, form an integral domain, usually written as $Z[i]$.

In this dissertation, Gaussian integers are denoted with capital letters and real integers with lower case letters. Also, vector notation for Gaussian integers is used (i.e., $G=(a,b)$ is equivalent to $G=a+bi$).

The multiplication of Gaussian integers is a case of complex number multiplication. If $G=(a,b)$ and $H=(c,d)$, then

$$GH = (a + bi)(c + di) = ac - bd + i(ad + bc) = (ac - bd, ad + bc) \tag{1.2}$$

Consequently,

$$G^2 = (a+bi)(a+bi) = a^2 - b^2 + i(2ab) = (a^2 - b^2, 2ab) \tag{1.3}$$

It takes three integer multiplications to multiply two Gaussian integers:

Algorithm 1.3.1 Multiplication of two Gaussian integers

Given: (a,b), (c,d) Gaussian integers

Find: Gaussian integer $(x,y) = (a,b)(c,d)$

$$v_1 = (a+b)(c+d); \tag{1.4}$$

$$v_2 = ac; \tag{1.5}$$

$$v_3 = bd \tag{1.6}$$

$$x = v_2 - v_3 \tag{1.7}$$

$$y = v_1 - v_2 - v_3 \tag{1.8}$$

Return (*x*,*y*)

It takes only two integer multiplications to square a Gaussian integer:

Algorithm 1.3.2 Squaring of two Gaussian integers

Given: (a,b) Gaussian integer

Find: Gaussian integer $(x,y)=(a,b)^2$

Return $(x,y)=(a,b)^2=\big((a+b)(a-b),ab+ab\big)$

The addition of Gaussian integers is a case of complex number addition. If G=(a,b) and H=(c,d), then

$$G+H=(a+bi)+(c+di)=a+c+i(b+d)=(a+c,b+d) \tag{1.9}$$

The norm of a Gaussian integer is the natural number defined as

$$|G|=|a+bi|=|(a,b)|=a^2+b^2 \tag{1.10}$$

It is known that $|GH|=|G||H|$ (by the properties of complex numbers).

All real integers are also Gaussian integers. The multiplication of a Gaussian integer by a real number is a case of the Gaussian integer multiplication:

If G=(a,b) is a Gaussian integer and h is a real integer, then:

$$Gh=(a+bi)h=ah+i(bh)=(ah,bh) \tag{1.11}$$

or equivalently:

$$Gh=(a+bi)(h+i\cdot 0)=(a,b)(h,0)=(ah,bh) \tag{1.12}$$

All real primes can be divided into two subgroups: primes p: p mod 4 = 3 and primes p: p mod 4 = 1. The primes p: p mod 4 = 3 will be referred to as Blum primes and primes p: p mod 4 = 1 as non-Blum primes.

The prime elements of $Z[i]$ are also known as Gaussian primes. If P is a Gaussian prime it cannot be represented as a product of non-unit Gaussian integers. The unit Gaussian integers are 1,-1, i and $-i$. Real prime numbers p: p mod 4 =3 are also Gaussian primes. Real prime numbers *p: p* mod 4 = 1 are not Gaussian primes since they can be represented as a sum of squares (according to the Fermat's theorem on sums of two squares) and, consequently, as a product of two Gaussian integers. For instance, $5 = 2^2 + 1^2 = (2+i)(2-i)$

Gaussian primes can be divided into two subgroups. One subgroup consists of primes $P=(p,0)$, where p is a real prime and p mod 4=3 or a real Blum prime. The second subgroup consists of primes $P=(a,b)$ where $|P|$ is a real prime and $|P|$ mod 4=1. The Gaussian primes $P=(p,0)$ will be referred as Blum Gaussian primes and the Gaussian primes $P=(a,b)$ where $|P|$ is a real prime will be referred as non-Blum Gaussian primes.

The division of Gaussian integers in this dissertation will be denoted as "DIV". It is analogous to integer division (commonly referred to as "div"). "DIV" operation may be defined in several ways. The most common two ways to define it is presented below. If G=(a,b) and H=(c,d) are Gaussian integers , then G DIV H can be defined as:

1)

$$G \text{ DIV } H = \left\lfloor \frac{ac+bd}{|H|} \right\rfloor + i \left\lfloor \frac{bc-ad}{|H|} \right\rfloor \tag{1.13}$$

2)

$$G \text{ DIV } H = \text{round}\left(\frac{ac+bd}{|H|}\right) + \text{round}\left(\frac{bc-ad}{|H|}\right), \tag{1.14}$$

where "round" operation is defined as

$$\text{round}(x) = \begin{cases} \lfloor x+0.5 \rfloor, x \geq 0 \\ \lceil x-0.5 \rceil, x < 0 \end{cases} \tag{1.15}$$

Modular congruence is defined over Gaussian integers in the similar way it is defined for real integers. If G=(a,b) and H=(c,d) are Gaussian integers then

$$G \text{ MOD } H = G - H\left(G \text{ DIV } H\right) \tag{1.16}$$

To differentiate Gaussian modulo operation from real integer modulo operation the notation "MOD" will be used to represent Gaussian modulo operation and "mod" will be used for real integer modulo operation.

Modular congruencies for Gaussian integers have similar properties as modular congruencies for real integers. However, there is an important difference: the residues modulo Gaussian primes are not unique. In fact, if $A \equiv B \text{ MOD } C$ then $A \equiv Bi \text{ MOD } C$, $A \equiv -Bi \text{ MOD } C$ and $A \equiv -B \text{ MOD } C$. Moreover, different ways to define division lead to different outcomes of Gaussian modulo operation. Regardless of the way the division is defined all the properties of modulo operation hold. When used for cryptography, the

non-unique outcomes of modulo operation present a problem. However, with consistent definitions of division this problem is overcome.

The G MOD H operation can be greatly simplified when H=(c,0) (or real integer). This operation will be defined as follows:

$$G \text{ MOD } H = (a,b)\text{MOD } (c,0) = G \bmod c = (a,b) \bmod c = (a \bmod c, b \bmod c), \qquad (1.17)$$

where G=(a,b) and H=(c,0) are Gaussian integers; $a \bmod c$ and $b \bmod c$ are regular real integer "mod" operations. This definition is consistent with the definition of modulo operation for Gaussian integers. Note the same "mod" notation is used to represent real integer modulo real integer operation and Gaussian integer modulo real integer operation. This does not cause inconsistencies because the real integer modulo operation can be looked at as a special case of Gaussian integer modulo real integer operation. If G=(a,0) and H=(c,0) are Gaussian integers and e=a mod c is a real integer, then

$$G \text{ MOD } H = (a,0) \bmod c = (e,0) \Leftrightarrow a \bmod c = e \qquad (1.18)$$

Below the formal definitions for modular operation on Gaussian integers are presented.

Definition 1.3.1 MOD Operation on Gaussian integers

If G and H are Gaussian integer, then

$$G \text{ MOD } H = G - H\left(G \text{ DIV } H\right) \qquad (1.19)$$

Definition 1.3.2 mod Operation on Gaussian integers

If G=(a,b) is a Gaussian integer c is a real integer, then

$$G \bmod c = (a,b) \bmod c = (a \bmod c, b \bmod c) \tag{1.20}$$

Note that Blum Gaussian primes are real primes so Definition 1.3.2 also applies.

The order for Gaussian integers is defined in the some way it is defined for real integers. Below is the formal definition of the order:

Definition 1.3.3 Order of a Gaussian integers

If H is a Gaussian integer, P is a Gaussian prime, k is a real integer, and $k > 1$, then k is referred to as the *order* of H (or ord(H) = k MOD P) if H^{k+1}=H (MOD P) and there is no such $m : 1 < m < k$ and $H^m = H$ MOD P .

If the Gaussian primes are restricted to Blum Gaussian primes, it is possible to define the order in terms of "mod" operation:

Definition 1.3.4 Order of a Blum Gaussian integers

If H is a Gaussian integer, p is a Blum Gaussian prime, k is a real integer , then k is referred to as the order of H (or ord(H) = k mod p) if H^{k+1}=H (mod p) and there is no such $m : 1 < m < k$ and $H^m = H \bmod p$.

Gaussian integer Discrete Logarithm Problem (DLP) is defined in the similar way the real integer DLP is defined. In the subsequent discussion, to differentiate between

these two problems, the Gaussian integer DLP will be denoted with “LOG” and the real integer DLP with “log”.

Definition 1.3.5 Gaussian integer discrete logarithm

If G and H are a Gaussian integers, P is a Gaussian Prime, k is a real integer and $G^k = H \pmod{P}$, or $\mathrm{LOG}_G H = k \pmod{P}$.

For Blum Gaussian primes DLP is defined as follows:

Definition 1.3.6 Gaussian integer discrete logarithm (Blum Gaussian primes)

If G and H are a Gaussian integers, p is a Blum Gaussian prime, k is a real integer and $G^k = H \pmod{p}$, then $\mathrm{LOG}_G H = k \pmod{p}$.

Note that a different notation for Gaussian DLP modulo Blum Gaussian primes is not required because it is differentiated by “MOD” vs. “mod” notation.

The notion of a generator for Gaussian integers is defined in the same way as for real integers. The formal definition is below:

Definition 1.3.7 Gaussian integer generator (Blum Gaussian primes)

A Gaussian integer G is a generator for a Blum Gaussian prime p iff $\mathrm{ord}(G) = p^2 - 1 \pmod{p}$.

Note that here a generator for non-Blum Gaussian primes is not defined. The reason for this is that such generators are not relevant to the subsequent discussion.

It is worth noting that Gaussian integers form a square lattice ([25]). Moreover, Gaussian integers are examples of a more general type of numbers: quadratic integers ([25]). It is possible to extend the results presented in this dissertation to quadratic integers as described in [25]. Specifically, it is possible to use *imaginary quadratic integers*:

$$\mathbb{Z}[\sqrt{r}] = \{a + b\sqrt{r} : a, b \in \mathbb{Z} \text{ and } r \text{ QNR}\} \tag{1.21}$$

Such generalization would allow for use of all real primes p (not just Blum primes) and still have the large order ($p^2 - 1$). In this dissertation, however, only Gaussian integers are considered (i.e., $\mathbb{Z}[\sqrt{-1}] = \{a + b\sqrt{-1} : a, b \in \mathbb{Z}\}$). In practice, this is not a significant restriction since it is very easy to find primes $p : p \bmod 4 = 3$.

1.4 Dissertation Structure

This dissertation contains five main chapters and conclusion. In this chapter, the notation and definitions were introduced along with the introduction and the survey of references. Chapter 2 is concerned with the Discrete Logarithm Problem (DLP) with Gaussian integers and the exponentiation of Gaussian integers. The main themes of Chapter 2 are the properties of the Gaussian integer exponentiation, comparisons of the Gaussian integer DLP to the real integer DLP and computational experiments confirming the theoretical findings. It is shown that the cryptosystems based on the Gaussian integer DLP have advantages over equivalent in security real integer cryptosystems. Moreover, a

novel algorithm for the Gaussian integer exponentiation (Algorithm 2.8.1, Lucas sequence Exponentiation of Gaussian integers (LSEG)) is introduced and its advantages proven theoretically and experimentally.

In Chapter 3 and Chapter 4, factoring based cryptosystems with Gaussian integers are discussed. In Chapter 3, an extension of Rabin cryptosystem into domain of Gaussian integers is introduced and discussed. The extension offers an advantage of using less reserved bits required for Rabin cryptosystem. In Chapter 4, various extensions or RSA into the field of Gaussian integers are analyzed. Some of the extensions are shown to be non-viable and for viable extensions it is hard to quantify any benefits over real integer RSA.

In Chapter 5, a new algorithm, designed to be used with most existing watermarking algorithms, is introduced. The new algorithm (Algorithm 5.2.1, Pixel rearrangement based on Gaussian integers) is based on the Gaussian integer exponentiation. The performance and benefits of this algorithm are discussed and compared with the existing algorithms.

After each chapter there is a short summary section. The last chapter (Chapter 6) is the overall conclusion.

CHAPTER 2

DISCRETE LOGARITHM CRYPTOGRAPHY WITH GAUSSIAN INTEGERS

2.1 Gaussian Primes *P*: |*P*| is a non-Blum Prime

Gaussian primes can be divided into two subgroups. One subgroup consists of primes $P=(p,0)$ where p is a real prime and p MOD 4=3 or real Blum primes. The second subgroup consists of primes $P=(a,b)$ where $|P|$ is a real prime and $|P|$ MOD 4=1. In this work, the first subset of Gaussian primes namely Blum primes is considered. In [27], this subset was also used to extend ElGamal algorithm.

There are good reasons for restricting Gaussian domain. Some of the reasons are efficiency and simplicity. The question arises: is there anything missed by considering only Blum primes? The answer is that nothing is gained by using non-Blum Gaussian primes to extend well-known cryptosystems. The reason for this is that, for non-Blum Gaussian primes P, there is one to one mapping between Gaussian integers modulo P and real integers modulo $|P|$. This means that it is easy to switch between the two representations. Below is a simple algorithm to convert Gaussian integers modulo P to real integers modulo $p = |P|$.

Algorithm 2.1.1 Convert Gaussian integer to real integer modulo non-Blum Gaussian prime

Given: $G=(a,b)$ is a Gaussian integer,

P a Gaussian prime such that $|P| = p$ is a real prime and p mod 4 =1

Find: real integer g

Step 1. Compute

$$s = \sqrt{-1} \bmod p \tag{2.1}$$

such that

$$s \text{ MOD } P = i \tag{2.2}$$

Step 2.

$$g = a + bs \bmod p \tag{2.3}$$

is the corresponding real number.

Algorithm 2.1.2 Convert Gaussian integer to real integer modulo non-Blum Gaussian prime

Given: g a real integer,

p a real prime, p mod 4=1

P a Gaussian prime such that $|P| = p$

Find: Gaussian integer G

Step 1. Compute $G=(g,0)$ MOD P

A lemma introduced below to prove the validity of Algorithm 2.1.1 and Algorithm 2.2.2.

Lemma 2.1.1

If $G = a + bi$ and $H = c + di$ are two Gaussian integers, P is a Gaussian prime, $|P|=p$ is a prime such that $p \bmod 4 = 1$, $s = \sqrt{-1} \bmod p$ (i.e., $s = i$ MOD P), ; $g = a + bs$ MOD p , $h = c + ds$ MOD p and k are real integers, then the following facts are true:

1)

$$g \text{ MOD } P = G \text{ and } h \text{ MOD } P = H \tag{2.4}$$

2)

$$g=h \text{ MOD } p <=> G=H \text{ MOD } P \tag{2.5}$$

3)

$$gh \text{ MOD } P = GH \text{ MOD } P \tag{2.6}$$

4)

$$g+h \text{ MOD } P = (G+H) \text{ MOD } P \tag{2.7}$$

5)

$$g^k=h \text{ MOD } p <=> G^k=H \text{ MOD } P \tag{2.8}$$

Proof:

1)

$$g \text{ MOD } P = (a+bs) \text{ MOD } P= a+bi \text{ MOD } P = G \text{ MOD } P \tag{2.9}$$

$$h \text{ MOD } P = (c+ds) \text{ MOD } P= c+di \text{ MOD } P = H \text{ MOD } P \tag{2.10}$$

2) Given $g=h$ MOD p.

$$a+bs= c+ds \quad (\text{MOD } p) \tag{2.11}$$

Appling (MOD P) operation to both sides of the equation:

$$a+bi=c+di \ \ (\text{MOD } P) \ \ => G=H \ (\text{MOD } P) \tag{2.12}$$

To prove the reverse assume that it is given that $G=H$ MOD P. Suppose $g \neq h \bmod p$.

$$a + bs \neq c + ds \ (\bmod p) \tag{2.13}$$

After applying (MOD P) operation to both sides of the equation:

$$a + bi \neq c + di \ (\text{MOD } P) \ => \ G \neq H \ (\text{MOD } P), \tag{2.14}$$

which is a contradiction because $G=H$ MOD P. Consequently,

$$g=h \text{ MOD } p. \tag{2.15}$$

3)

$$\begin{aligned} &gh \text{ MOD } P = (a + bs)(c + ds) \text{ MOD } P = \\ &= ac + bds^2 + s(bc + ad) \text{ MOD } P = \\ &= (ac - bd) + i(bc + ad) \text{ MOD } P = GH \text{ MOD } P \end{aligned} \tag{2.16}$$

4)

$$g+h \text{ MOD } P = (a+bs+c+ds) \text{ MOD } P = a+c+s(b+d) \text{ MOD } P = \\ = a+c+i(b+d) \text{ MOD } P = G+H \text{ MOD } P \quad (2.17)$$

5) Given $g^k = h \bmod p$, or:

$$(a+bs)^k = h \quad (\text{MOD } p) \quad (2.18)$$

Applying (MOD P) operation to both sides of the equation:

$$(a+bs)^k \text{ MOD } P = h \text{ MOD } P \quad (2.19)$$

$$((a+bs) \text{ MOD } P)^k \text{ MOD } P = h \text{ MOD } P \quad (2.20)$$

$$((a+bi) \text{ MOD } P)^k \text{ MOD } P = H \text{ MOD } P \quad (2.21)$$

$$G^k = H \text{ MOD } P \quad (2.22)$$

To prove the reverse, assume that it is given that $G^k = H$ MOD P. Suppose $g^k \neq h \bmod p$, then:

$$(a+bs)^k \neq h \bmod p \quad (2.23)$$

Applying (MOD P) operation to both sides of the equation:

$$(a+bs \text{ MOD } P)^k \text{ MOD } P \neq h \text{ MOD } P \tag{2.24}$$

$$(a+bi)^k \text{ MOD } P \neq h \text{ MOD } P \tag{2.25}$$

$$G^k \neq H \text{ MOD } P, \tag{2.26}$$

which is a contradiction, thus?

$$g^k = h \bmod p \tag{2.27}$$

Q.E.D.

Lemma 2.1.1 implies that DLP problem for Gaussian integers modulo non-Blum Gaussian primes can be solved using real integers. An example below illustrates this point:

Example 2.1.1 Reduction of the Gaussian integer DLP modulo non-Blum Gaussian prime to the real integer DLP

Given: $P = 3+2i$, $|P| = p = |3+2i| = 13$.

$G = 1+i$,

$G^k = 1-i$ MOD $(3+2i)$

Find: Need to find k.

Solution:

Using Lemma 2.1.1 $G^k=H$ MOD P <=> $g^k=h$ MOD p.

For p=13, $\sqrt{-1} = \sqrt{p-1} = \sqrt{12} \bmod 13$. There are two square roots of –1 MOD 13: 5 and 8.

5^2 MOD 13=12 and $8^2 \bmod 13 = 12$

However, 8 MOD (3+2i) =-i and 5 MOD (3+2i) =i so set s=5.

$g = 1+s = 1+5 = 6 \pmod{13}$

$h = 1-s = 1-5 = -4 = 9 \pmod{13}$

In order to find k, the real integer DLP needs to be solved:

$6^k = 9$ MOD 13

The solution is $k = 4$. Indeed, $(1+i)^4$ MOD (3+2i) = 1-i.

Example 2.1.1 illustrates how DLP problem for Gaussian integers is reduced to the real integer DLP problem. This implies that using Gaussian integers modulo non-Blum Gaussian primes for DLP type cryptosystems does not give any advantages over the real integers algorithms. It introduces complexity without any apparent advantages.

Table 2.1 below show discrete power Gaussian integer groups MOD P and the corresponding real integer group MOD $|P|$.

Table 2.1 Discrete Power Table MOD P=3+2i, $|P|$=13, $\sqrt{-1} \bmod 13 = 5$ [1]

g	(G^1) [g^1]	(G^2) [g^2]	(G^3) [g^3]	(G^4) [g^4]	(G^5) [g^5]	(G^6) [g^6]	(G^7) [g^7]	(G^8) [g^8]	(G^9) [g^9]	(G^{10}) [g^{10}]	(G^{11}) [g^{11}]	(G^{12}) [g^{12}]
1	(1) [1]											
2	(2) [2]	(-1+i) [4]	(-i) [8]	(-2i) [3]	(1+i) [6]	(-1) [12]	(-2) [11]	(1-i) [9]	(i) [5]	(2i) [10]	(-1-i) [7]	(1) [1]
3	(-2i) [3]	(1-i) [9]	(1) [1]									
4	(-1+i) [4]	(-2i) [3]	(-1) [12]	(1-i) [9]	(2i) [10]	(1) [1]						
5	(i) [5]	(-1) [12]	(-i) [8]	(1) [1]								
6	(1+i) [6]	(2i) [10]	(-i) [8]	(1-i) [9]	(2) [2]	(-1) [12]	(-1-i) [7]	(-2i) [3]	(i) [5]	(-1+i) [4]	(-2) [11]	(1) [1]
7	(-1-i) [7]	(2i) [10]	(i) [5]	(1-i) [9]	(-2) [11]	(-1) [12]	(1+i) [6]	(-2i) [3]	(-i) [8]	(-1+i) [4]	(2) [2]	(1) [1]
8	(-i) [8]	(-1) [12]	(i) [5]	(1) [1]								
9	(1-i) [9]	(-2i) [3]	(1) [1]									
10	(2i) [10]	(1-i) [9]	(-1) [12]	(-2i) [3]	(-1+i) [4]	(1) [1]						
11	(-2) [11]	(-1+i) [4]	(i) [5]	(-2i) [3]	(-1-i) [7]	(-1) [12]	(2) [2]	(1-i) [9]	(-i) [8]	(2i) [10]	(1+i) [6]	(1) [1]
12	(-1) [12]	(1) [1]										

[1] Gaussian integers are shown in (). The corresponding real integers are shown in [].

Table 2.1 illustrates the one to one correspondence between Gaussian integers modulo non-Blum Gaussian primes and real integers. It also illustrates that exponentiation operation is also equivalent.

As it was shown, the Gaussian integers modulo non-Blum Gaussian primes are equivalent to real primes as far as DLP problem is concerned. For this reason such primes are excluded from the further DLP analysis which focuses on Blum Gaussian primes.

2.2 Common Cryptography Algorithms Based on Discrete Logarithm

Gaussian integers can replace real integers in cryptosystems that are based on the difficulty of computing the Discrete Logarithm. Two most common of these cryptosystems are the Diffie-Hellman Key Exchange protocol and the ElGamal algorithm.

In 1976, Diffie and Hellman introduced a new key exchange algorithm. This algorithm is still widely used.

Algorithm 2.2.1 The original Diffie-Hellman Key Exchange protocol

1. Alice and Bob agree to use a prime number p and a generator g.
2. Alice chooses a secret integer a: $1 < a < p\text{-}1$, computes

$$g^a \bmod p \tag{2.28}$$

   and sends the result to Bob.
3. Bob chooses a secret integer b: $1 < b < p\text{-}1$, computes

$$g^b \bmod p \tag{2.29}$$

and sends the result to Alice.

4. Alice computes the shared key as follows

$$k = (g^b \bmod p)^a \bmod p \tag{2.30}$$

5. Bob computes the shared key as follows

$$k = (g^a \bmod p)^b \bmod p \tag{2.31}$$

In 1984, Taher ElGamal introduced ElGamal algorithm.

Algorithm 2.2.2 ElGamal algorithm over the field of real integers

Key generation

- Alice and Bob agree on a prime p and a generator g.
- Alice generates a secret integer a: $1 < a < p$-1 and computes her private key

$$k_a = g^a \bmod p \tag{2.32}$$

- Bob generates a secret integer b: $1 < b < p$-1 and computes his private key

$$k_b = g^b \bmod p \tag{2.33}$$

Encryption (Bob's actions)

- Bob selects a random integer $1 < s < p\text{-}1$.
- Given message m: $0 \leq m \leq n - 1$ Bob computes the ciphertext

$$c = m(k_a)^S \bmod p \tag{2.34}$$

- Bob computes hint

$$h = g^S \bmod p \tag{2.35}$$

- Bob sends both c and h to Alice

Decryption (Alice's actions)

- Alice computes

$$m = ch^{-a} \bmod p \tag{2.36}$$

Extending the Diffie-Hellman Key Exchange protocol is straightforward. The extended algorithm is below:

Algorithm 2.2.3 Diffie-Hellman Key Exchange protocol over the field of Gaussian integers

1. Alice and Bob agree to use a prime number p and a Gaussian integer generator G.
2. Alice chooses a secret integer a: $1 < a < p^2\text{-}1$, computes

$$G^a \bmod p \tag{2.37}$$

and sends the result to Bob.

3. Bob chooses a secret integer b: $1 < b < p^2$-1, computes

$$G^b \bmod p \tag{2.38}$$

and sends the result to Alice.

4. Alice computes the shared key as follows

$$K=(\ G^b \bmod p)^a \bmod p. \tag{2.39}$$

K is a Gaussian integer.

5. Bob computes the shared key as follows

$$K=(\ G^a \bmod p)^b \bmod p \tag{2.40}$$

It is also quite easy to extend ElGamal algorithm into the field of Gaussian integers. Such an extension is described in [27]:

Algorithm 2.2.4 ElGamal algorithm over the field of Gaussian integers

Key generation

- Alice and Bob agree on a prime p and a Gaussian integer generator G.
- Alice generates a secret integer a: $1 < a < p^2$-1 and computes her private key

$$K_a = G^a \bmod p\,, \tag{2.41}$$

K_a is a Gaussian integer.

- Bob generates a secret integer b**:** $1<b<p^2-1$ and computes his private key

$$K_b = G^b \bmod p \tag{2.42}$$

K_b is a Gaussian integer.

Encryption (Bob's actions)

- Bob selects a random integer $1 < s < p^2-1$.
- Given message M , Bob computes the ciphertext

$$C = M(K_a)^S \bmod p \tag{2.43}$$

M, K_a and C are Gaussian integers.

- Bob computes hint

$$H = G^S \bmod p \tag{2.44}$$

H is a Gaussian integer.

- Bob sends both C and H to Alice

Decryption (Alice's actions)

- Alice computes

$$M = CH^{(p^2-a)} \bmod p \tag{2.45}$$

2.3 Properties of Gaussian Integer Exponentiation

For any two complex numbers A and B it is true that $|AB|=|A||B|$. Gaussian integer is a special kind of complex number so it is true for Gaussian integers also. When a Gaussian integer C is multiplied by itself modulo p, in turn, the norm of $C \bmod p$ gets multiplied by itself also. This means that $\left|C^i\right| \bmod p \ (i=1,2,...)$ will cycle with a period of $\text{ord}\left(\left|C\right|\right) \bmod p$ as illustrated in examples below.

Table 2.2 Repeating Norm Example for Prime ***p***=7

Power:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Norm:	2	4	**1**	2	4	**1**	2	4	**1**	2	4	**1**	2	4	**1**	
	(1,6) [2]	(0,5) [4]	**(5,5) [1]**	(3,0) [2]	(3,4) [4]	**(0,1) [1]**	(1,1) [2]	(2,0) [4]	**(2,5) [1]**	(0,3) [2]	(3,3) [4]	**(6,0) [1]**	(6,1) [2]	(0,2) [4]	**(2,2) [1]**	(4,0) [2]
	(1,1) [2]	(0,2) [4]	**(5,2) [1]**	(3,0) [2]	(3,3) [4]	**(0,6) [1]**	(1,6) [2]	(2,0) [4]	**(2,2) [1]**	(0,4) [2]	(3,4) [4]	**(6,0) [1]**	(6,6) [2]	(0,5) [4]	**(2,5) [1]**	(4,0) [2]
Norm:	3	2	6	4	5	**1**	3	2	6	4	5	**1**	3	2	6	4
	(3,1) [3]	(1,6) [2]	(4,5) [6]	(0,5) [4]	(2,1) [5]	**(5,5) [1]**	(3,6) [3]	(3,0) [2]	(2,3) [6]	(3,4) [4]	(5,1) [5]	**(0,1) [1]**	(6,3) [3]	(1,1) [2]	(2,4) [6]	(2,0) [4]
	(4,6) [3]	(1,6) [2]	(3,2) [6]	(0,5) [4]	(5,6) [5]	**(5,5) [1]**	(4,1) [3]	(3,0) [2]	(5,4) [6]	(3,4) [4]	(2,6) [5]	**(0,1) [1]**	(1,4) [3]	(1,1) [2]	(5,3) [6]	(2,0) [4]

Table 2.3 Repeating Norm Examples for Prime p=11

Power:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Norm:	2	4	8	5	10	9	7	3	6	**1**	2	4	8	5	10
	(3,2) [2]	(5,1) [4]	(2,2) [8]	(2,10) [5]	(8,1) [10]	(0,8) [9]	(6,2) [7]	(3,7) [3]	(6,5) [6]	**(8,5) [1]**	(3,9) [2]	(2,0) [4]	(6,4) [8]	(10,2) [5]	(4,4) [10]
	(10,1) [2]	(0,9) [4]	(2,2) [8]	(7,0) [5]	(4,7) [10]	(0,8) [9]	(3,3) [7]	(5,0) [3]	(6,5) [6]	**(0,1) [1]**	(10,10) [2]	(2,0) [4]	(9,2) [8]	(0,7) [5]	(4,4) [10]
Norm:	3	9	5	4	**1**	3	9	5	4	**1**	3	9	5	4	**1**
	(3,4) [3]	(4,2) [9]	(4,0) [5]	(1,5) [4]	**(5,8) [1]**	(5,0) [3]	(4,9) [9]	(9,10) [5]	(9,0) [4]	**(5,3) [1]**	(3,7) [3]	(3,0) [9]	(9,1) [5]	(1,6) [4]	**(1,0) [1]**
	(7,8) [3]	(7,2) [9]	(0,4) [5]	(1,6) [4]	**(3,6) [1]**	(6,0) [3]	(9,4) [9]	(9,1) [5]	(0,2) [4]	**(6,3) [1]**	(7,3) [3]	(3,0) [9]	(10,2) [5]	(10,6) [4]	**(0,1) [1]**

In addition, $C^{\text{ord}(|C|)}$ is a Gaussian integer with norm equal to 1 mod p. In fact, the Gaussian integers U: $|U| = 1 \bmod p$ form a cyclic subgroup with an order $(p+1)$. This subgroup will be referred as a *Norm 1* subgroup. Moreover, order of any Gaussian integer C is a product of ord($|C|$) and ord($|U|$) where $U = C^{\text{ord}(|C|)} \bmod p$. The algorithms for finding Gaussian Generators to use for Discrete Logarithm based cryptography are derived from this. The series of theorems below will prove these facts formally.

Lemma 2.3.1

If C is a complex number, p is a prime, then

$$|C^n| = |C|^n \bmod p \tag{2.46}$$

Proof:

For any complex number it is true that $|C^n| = |C|^n$, therefore $|C^n| = |C|^n \bmod p$

Q.E.D

Lemma 2.3.2

If C is a Gaussian integer, p is a Blum prime

1) ord(C) mod p is divisible by ord(|C|) mod p
2) if $C^{\text{ord}(|C|)} = U \bmod p$, then $|U| = 1 \bmod p$
3) if $U = C^{\text{ord}(|C|)} \bmod p$, then ord($C$) mod p is divisible by ord(U) mod p

Proof:

1) Suppose ord(C) mod p is not divisible by ord(|C|) mod p. This would mean that $|C^{\text{ord}(C)}|$ mod p is not equal to 1 but $C^{\text{ord}(C)} = (1,0)$. This is a contradiction.
2) $|U|$ must equal to 1 mod p because $|C^n| = |C|^n \bmod p$ and, in this case, $n = \text{ord}(|C|)$.
3) If ord(C) mod p is not divisible by ord(U) then $C^{\text{ord}(C)}$ would not equal to (1,0) so ord(C) must be divisible by ord(U).

Q.E.D

Lemma 2.3.3

If U is a Gaussian integer, p is a Blum prime and $|U| = 1 \bmod p$

1) the maximum order of U is (p+1)
2) ord(U) mod p must divide (p+1)

Proof:

1) Any Gaussian integer A taken to the power (p+1) mod p is in the form (c,0).

In this case, $U^{p+1} \bmod p$ could be one of either (1,0) or (-1,0) because $|U| = 1 \bmod p$. Since p+1 is divisible by 4 for all Blum primes, $U^{(p+1)/4}$ is a

Gaussian integer of *Norm 1* and is a root of degree of U^{p+1}. For (-1,0) all roots of degree four have a norm equal to –1 mod p. This means that $U^{(p+1)}$ must equal to (1,0) mod p.

2) If p+1 is not divisible by ord(U) then $U^{(p+1)}$ would not equal to (1,0) so p+1 must be divisible by ord(U).

Q.E.D.

Lemma 2.3.4

If C is a Gaussian integer, p is a Blum prime then

$$\text{ord}(C) = \text{ord}(|C|)\text{ord}(C^{\text{ord}(|C|)}) \bmod p\,, \tag{2.47}$$

Proof:

ord(C) must be divisible by ord(|C|) and ord(U) so

$$\text{ord}(C) = n\text{ord}(|C|)\text{ord}(U), \tag{2.48}$$

where n is an integer. In addition,

$$C^{\text{ord}(|C|)\text{ord}(U)} = U^{\text{ord}(U)} = (1,0). \tag{2.49}$$

Consequently, n must equal to 1.

Q.E.D.

Lemma 2.3.5 The order of Gaussian integers U' where $|U'| = -1$

If U is a Gaussian integer, p is a Blum prime and $|U'|=-1 \bmod p$

3) the maximum order of U' is $2(p+1)$

4) $\text{ord}(U') \bmod p$ must divide $2(p+1)$

Proof:

The proof follows directly from Lemma 2.3.3. Note that Gaussian integers U in Lemma 2.3.3 are squares of U' mod p. Therefore, 1) and 2) must be true.

Q.E.D.

Corollary 2.3.1

Let C be a Gaussian integer, p a Blum prime. (p^2-1) is divisible by $\text{ord}(C) \bmod p$.

Corollary 2.3.2

Let G be a Gaussian integer, p a Blum prime. G is a generator if and only if

$$\text{ord}(|G|) = p-1 \tag{2.50}$$

and

$$\text{ord}(U) \bmod p = p+1, \tag{2.51}$$

where

$$U = G^{p-1} \bmod p, \tag{2.52}$$

Corollary 2.3.1 validates Algorithm 2.10.1. In Algorithm 2.10.1, all the possible powers of G mod p that can possibly equal to (1,0) are tested. If G is a generator, only $G^{(p^2-1)} \bmod p$ equals to (1,0).

2.4 Discrete Logarithm Complexity for Gaussian Integers

When using Gaussian integers for Discrete Logarithm based cryptography, an important question arises, namely, is it secure? In [38], the Pollard Rho algorithm attack was used to asses the security of ElGamal algorithm with Gaussian integers. The results were encouraging: it took twice as much time to compute discrete logarithm for Gaussian integers. However, these results do not prove that it is secure. In this section, the problem of Discrete Logarithm Problem (DLP) for Gaussian integers is analyzed.

It is clear that DLP for Gaussian integers is at least as hard as DLP for real integers, because real integers are a special case of Gaussian integers. Another way of stating this is that whenever the DLP for Gaussian integer G modulo p is solved, the real integer DLP for the norm of G is also solved. This means that DLP for Gaussian integers is at least as hard as DLP for real integers, thus, the Gaussian integer DLP cryptography is at least as secure as the analogous real integer DLP cryptography.

Suppose

$$G^k \bmod p = C \tag{2.53}$$

is given, where G and C are Gaussian integers. k can be represented as

$$k = su + r\,, \tag{2.54}$$

where u = ord($|G|$) mod p, and $r = k \bmod u$. If $|G|$ is a real generator, then u equals to p-1. In order to find $\mathrm{LOG}_G(C) \bmod p$, it is sufficient to find s and r. Since ord($|G|$) mod p divides p-1 it can be assumed that

$$u = p - 1 \tag{2.55}$$

for any G.

The problem of finding r is the well known real integer DLP problem:

$$r = \log_{|G|}(|C|) \bmod p\,, \tag{2.56}$$

Once r is known, s has to be found. Suppose a Gaussian integer D is

$$D = C(G^{-1})^r \bmod p\,, \tag{2.57}$$

and a Gaussian integer U is

$$U = G^{\mathrm{ord}(|G|)} \bmod p\,, \tag{2.58}$$

According to Lemma 2.3.2, $|U| = |D| = 1 \bmod p$. In order to find s, it is necessary to find

$$s = \mathrm{LOG}_U(D) \bmod p \tag{2.59}$$

Both U and D belong to the *Norm 1* subgroup, because according to Lemma 2.3.2:

$$|U| = |D| = 1 \bmod p. \tag{2.60}$$

Example 2.4.1 Computing the Gaussian integer DLP in $\mathrm{O}(\sqrt{p})$

Using

Table 2.3**,** suppose the task is to compute $\mathrm{LOG}_{(3,4)}(9,1) \bmod 11$. Here $G = (3,4)$; $C = (9,1)$; $p = 11$; $|G| \bmod p = 3 \bmod 11$; $|C| \bmod p = 5 \bmod 11$; $G^{-1} \bmod p = (1,6) \bmod 11$; $u = \mathrm{ord}(|G|) \bmod p = 5$.

First find r:

$$r = \log_{|G|}(|C|) \bmod p = \log_3(5) \bmod 11 = 3, \tag{2.61}$$

Now find D and U:

$$D = C \cdot (G^{-1})^r \bmod p = (9,1) \cdot (1,6)^3 \bmod 11 = (5,3) \bmod 11, \tag{2.62}$$

$$U = G^{\mathrm{ord}(|G|)} \bmod p = (3,4)^5 \bmod 11 = (5,8) \bmod 11, \tag{2.63}$$

Note that:

$$|D| = |(5,3)| = 5^2 + 3^2 = 34 = 1 (\bmod 11), \tag{2.64}$$

and

$$|U| = |(5,8)| = 5^2 + 8^2 = 89 = 1 (\bmod 11). \tag{2.65}$$

Now find s:

$$s = \mathrm{LOG}_U(D) \bmod p = \mathrm{LOG}_{(5,8)}(5,3) \bmod 11 = 2. \tag{2.66}$$

To find $k = \mathrm{LOG}_{(3,4)}(9,1) \bmod 11$:

$$k = su + r = 2 \cdot 5 + 3 = 13 \tag{2.67}$$

Indeed, $(3,4)^{13} \bmod 11 = (9,1)$.

The problem (2.66) is different from the real integer DLP. The solution to this problem is the key to understanding of how much complexity Gaussian integer extension adds to DLP. One way to solve it is to use any general DLP algorithm for a cyclic group, such as Baby-step giant-step or Pollard's Rho algorithm. These algorithms work for any cyclic group and their running time is $O(\sqrt{N})$, where N is the order of this cyclic group. According to Lemma 2.3.3, the order of *Norm 1* subgroup modulo prime p is p+1. Thus, s can be computed in $O(\sqrt{p+1}) = O(\sqrt{p})$ operations. Consequently, it is possible to compute Gaussian integer discrete logarithm with $O\left(\sqrt{p}\right)$, because the running time for solving (2.5.4) is $O(\sqrt{p-1}) = O(\sqrt{p})$. The combined running time for solving the Gaussian integer DLP is

$$O(\sqrt{p+1}) + O(\sqrt{p-1}) = O(\sqrt{p}). \tag{2.68}$$

In [36] and [38], Pollard's Rho Algorithm and Baby-step giant-step were used to compute DLP for Gaussian integers. Both algorithms have the average running time of $O(\sqrt{n})$ where n is the order of the cyclic group. For Gaussian integers $n=p^2-1$ ([22]). Consequently, the expected running time for DLP attack, as described in [36] and [38], is

$$O(\sqrt{p^2-1}) = O(\sqrt{p^2}) = O(p). \tag{2.69}$$

The complexity of solution of the DLP can be further reduced if the generalized BSGS algorithm [64] is applied. The paper demonstrates several enhancements of the traditional Shank's algorithm.

Currently, the fastest algorithm for real integer Discrete Logarithm runs in sub-exponential time, which is substantially faster than $O(\sqrt{p})$. The apparent question arises: is there such an algorithm for the *Norm 1* subgroup? The answer to this question is the key to understanding of how much complexity Gaussian integer extension adds to DLP. In the next section, this question is addressed.

2.5 Reducing Gaussian Integer DLP to Lucas Sequences DLP

The Gaussian integer exponentiation operation can be represented as a recurrence. This representation is useful to derive and prove several formulas. It is useful to illustrate and prove properties that are not easily seen with other representations. Let $C=(a,b)$ be a Gaussian integer and $C^k=(a,b)^k=(a_k,b_k)$, then

$$C^0 = (1,0) = (a_0,b_0), \tag{2.70}$$

$$C^1 = (a,b) = (a_1,b_1), \tag{2.71}$$

$$C^2 = (a^2\text{-}b^2,2ab) = (a_2,b_2), \ \ldots \ C^k = (a,b)^k = (a_k,b_k). \tag{2.72}$$

The two dimensions of C are the recurrences a_i and b_i with an interesting property, described in the following theorem:

Theorem 2.5.1

Gaussian integer C^k mod p can be represented as a recurrence:

$$C^k = (a_k, b_k) = (2aa_{k-1} - |C| a_{k-2}, 2ab_{k-1} - |C| b_{k-2}) . \tag{2.73}$$

where a_0=1, a_1=a, b_0=0, b_1=b.

Proof: The theorem can be easily proved using the mathematical induction. The induction base:

$$C^2 = \left(a^2 - b^2, 2ab\right) = \left(2aa_1 - |C| a_0, 2ab_1 - |C| b_0\right) \tag{2.74}$$

is a true identity, since $a_0 = 1$, $a_1 = a$, $b_0 = 0$, $b_1 = b$.

Assume that for $r \le k$, the recurrence is true. The following needs to be proved:

$$C^k = (a_k, b_k) = \left(2aa_{k-1} - |C| a_{k-2}, 2ab_{k-1} - |C| b_{k-2}\right) \tag{2.75}$$

Using,

$$C^2 = (a_2, b_2) = (2a^2 - |C|, 2ab) , \tag{2.76}$$

$$C^k = \left(a_k, b_k\right) = C^{k-2}C^2 = \tag{2.77}$$

$$= \left(a_{k-2}a_2 - b_{k-2}b_2, a_{k-2}b_2 + b_{k-2}a_2\right) \tag{2.78}$$

$$a_k = a_{k-2}a_2 - b_{k-2}b_2 = \tag{2.79}$$

$$= a_{k-2}(2a^2 - |C|) - b_{k-2}2ab = \tag{2.80}$$

$$= a_{k-2}2a^2 - b_{k-2}2ab - |C|a_{k-2} = . \tag{2.81}$$

$$= 2a\left(aa_{k-2} - b_{k-2}b\right) - |C|a_{k-2} = 2aa_{k-1} - |C|a_{k-2} \tag{2.82}$$

$$b_k = a_{k-2}b_2 + b_{k-2}a_2 = a_{k-2}2ab + b_{k-2}\left(2a^2 - |C|\right) = \tag{2.83}$$

$$= a_{k-2}2ab + b_{k-2}2a^2 - b_{k-2}|C| = . \tag{2.84}$$

$$= 2a(a_{k-2}b + b_{k-2}a) - b_{k-2}|C| = 2ab_{k-1} - |C|b_{k-2} . \tag{2.85}$$

Q.E.D.

Using this theorem it is possible to show how to reduce the Gaussian integer DLP to the Lucas sequences DLP. The theorem below describes this relationship.

Theorem 2.5.2

If $C=(a,b)$ is a Gaussian integer, $C^k=(a_k,b_k)$, then the sequence $a_0,a_1,\ldots,a_k$ can be represented as a standard Lucas sequence $V\left(2a,|C|\right)$ as follows:

$$V_i\left(2a,|C|\right)=2a_i,\quad i=0,1,.... \tag{2.86}$$

and the sequence $b_0,b_1,\ldots,b_k$ relates to the Lucas sequence $U\left(2a,|C|\right)$ as follows:

$$U_i\left(2a,|C|\right)=b_ib^{-1},\quad i=0,1,.....\tag{2.87}$$

Proof: Using mathematical induction:

1) The theorem is correct for k=0 and k=1:

$$a_0=a^0=1\Rightarrow V_0=2=2a_0 \tag{2.88}$$

$$a_1=a^1=a\Rightarrow V_1=2a=2a_1. \tag{2.89}$$

$$b_0=0\Rightarrow U_0=0=0b^{-1}. \tag{2.90}$$

$$b_1 = b^1 = b \Rightarrow V_1 = 1 = b_1 b^{-1}. \tag{2.91}$$

2) Assume $V_i(2a,|C|) = 2a_i$ and $U_k(2a,|C|) = b_k b^{-1}$ for $i < k$.

3) Prove $V_k\left(2a,|C|\right) = 2a_k$ and $U_k\left(2a,|C|\right) = b_k b^{-1}$. According to Theorem 2.5.2:

$$a_k = 2a_{k-1} - |C| a_{k-2}. \tag{2.92}$$

$$b_k = 2b_{k-1} - |C| b_{k-2}. \tag{2.93}$$

Using the assumption of step 2) of the induction:

$$a_k = 2aa_{k-1} - |C| a_{k-2} = aV_{k-1}\left(2a,|C|\right) - 2^{-1}|C| V_{k-2}\left(2a,|C|\right) \tag{2.94}$$

$$2a_k = 2aV_{k-1}\left(2a,|C|\right) - |C| V_{k-2}\left(2a,|C|\right) = V_k\left(2a,|C|\right) \tag{2.95}$$

$$b_k = 2ab_{k-1} - |C| b_{k-2} = 2abU_{k-1}\left(2a,|C|\right) - b|C| U_{k-2}\left(2a,|C|\right) \tag{2.96}$$

$$b_k b^{-1} = 2aU_{k-1}\left(2a,|C|\right) - |C| U_{k-2}\left(2a,|C|\right) = U_k\left(2a,|C|\right). \tag{2.97}$$

Q.E.D.

In other words, whenever the Gaussian integer $C=(a,b)$ is raised to some power, the first dimension contains the Lucas sequence V_k and the second dimension contains the Lucas sequence U_k. The table below illustrates this.

Table 2.4 Gaussian Integer Exponentiation and Lucas Sequences Side-by-side

Power k:	0	1	2	3	4	5	6	7	8	9	10	11	12
$(3,7)^k$ mod 19	(1,0)	(3,7)	(17,4)	(4,17)	(7,3)	(0,1)	(12,3)	(15,17)	(2,4)	(16,7)	(18,0)	(16,12)	(2,15)
$2a_k$ mod 19	2	6	15	8	14	0	5	11	4	13	17	13	4
$V_k(6,1)$ mod 19	2	6	15	8	14	0	5	11	4	13	17	13	4
$b_k b^{-1}$ mod 19	0	1	6	16	14	11	14	16	6	1	0	18	13
$U_k(6,1)$ mod 19	0	1	6	16	14	11	14	16	6	1	0	18	13

The relationship described by Theorem 2.5.2 allows to reduce the Gaussian integer DLP to a better-known problem of the Lucas sequences DLP. Two cryptosystems based on DLP for Lucas sequences LUCDIF and LUCELG were introduced in [63]. These are Diffie-Hellman and ElGamal algorithms formulated with Lucas sequences $V_k(P,Q)$ where $Q \equiv 1 \bmod p$. The main selling point of "LUC" algorithms was a notion that they are not based on exponentiation; therefore, presumably, they are not vulnerable to sub-exponential time attacks. However, these assumptions were proven to be wrong. Papers [52] and [51] show that sub-exponential time algorithms can be applied to Lucas sequences.

On the other hand, Lucas sequences still offer a significant security advantage over real integers. The sub-exponential algorithm for Lucas sequences would have to run in a group of order p^2-1, as opposed the group of order of order p-1 with real integers.

Another important point to note is that *Norm 1* subgroup described in previous section contains Lucas sequences $V_k(P,Q)$ and $U_k(P,Q)$ with $Q \equiv 1 \bmod p$ (according to Theorem 2.5.2). Consequently, even though the order of *Norm 1* subgroup is p+1, the sub-exponential DLP algorithm would have to be applied in a group of order p^2-1. This means that the Gaussian integer DLP is substantially harder then the real integer DLP (with algorithms currently known). Moreover, when solving the Gaussian integer DLP one would have to solve two problems:

1. The Lucas sequences DLP with $Q \equiv 1 \bmod p$ (or equivalently the Gaussian integer DLP in the *Norm 1* subgroup).
2. The real integer DLP.

The fact that these two problems seem to be very different, bodes well for cryptography algorithms based on the Gaussian integer DLP. A solution of one problem may not lead to a solution of the other, thus the Gaussian integers offer additional protection.

When comparing the speed of the algorithms that utilize the DLP over Lucas sequences with the corresponding algorithms that utilize Gaussian integers, a strong case could be made for Gaussian integers even though this topic is not in scope of this dissertation. Nevertheless, the indications are that the Gaussian integer exponentiation is not slower than Lucas sequence computation under LUC cryptographic algorithms, but is

most likely faster. The actual speed varies greatly with implementation details and the choice of exponentiation algorithms.

The Gaussian integer DLP is a combination of two problems: the well known real integer DLP and the DLP over an interesting subgroup of Gaussian integer group: *Norm 1* subgroup. The complexity of *Norm 1* subgroup DLP holds the key to the understanding of the complexity of the Gaussian integer DLP. The reduction of the Gaussian integer DLP to the Lucas sequences DLP allows to assess the security of the Gaussian integer DLP. The *Norm 1* DLP turned out to be equivalent in security to the Lucas sequences DLP used in well-known cryptosystems LUCDIF and LUCELG. LUCDIF and LUCELG are thought to be more secure than the corresponding algorithms with real integers. Therefore, the algorithms based on the Gaussian integer DLP (such as the one in [27]) are more secure. Furthermore, the Gaussian integer DLP contains the real integer DLP, providing additional security through diversification. Luckily, algorithms with Gaussian integers are efficient and easy to implement. Moreover, they are potentially more efficient than the corresponding "LUC" algorithms. Thus, the algorithms based on the Gaussian integer DLP offer a great alternative to the real integer DLP or "LUC" algorithms.

2.6 Multiplication of Gaussian Integers vs. Real Integer Multiplication

Since the size of the group of the exponentiation cyclic group of Gaussian integers is $p^2 - 1$, it is appropriate to compare the Gaussian integer multiplication modulo p to real integer multiplication modulo q, where q is double the size of p. Suppose $n = \lceil \log_2 p \rceil$

(or in other words n is the number of binary bits of p). Let q be the closest prime to p^2. In that case, the size of q in bits would be approximately $2n$ or

$$\lceil \log_2 q \rceil \approx 2n . \tag{2.98}$$

For small n, the naïve multiplication method [44] with complexity of $\Theta(n^2)$ is most efficient. For larger n, the Karatsuba-Ofman [43] multiplication algorithm is appropriate with the running time of $\Theta(n^{1.585})$. For even larger n, the Toom-Cook 3-way (or Toom-3) [44] multiplication with the running time $\Theta(n^{1.465})$ is appropriate. As n increases further, multiple levels of the k-way Toom-Cook multiplication with the running time $\mathrm{O}\left(n^{\log(2k-1)/\log k}\right)$ can be applied ([44]). For extremely large n, the algorithms based on the Fast Fourier transforms (FFT) are more efficient. The Schönhage–Strassen algorithm [60] is based on the FFT and runs in $\mathrm{O}(n \log n \log\log n)$. The Fürer's algorithm published in [31] is also based on the FFT and offer an even better running time, that is $\mathrm{O}(n \log n 2^{\log^* n})$. The FFT algorithms have a lot of overhead, and, consequently are used for very large n, far larger then the numbers used for public key cryptography. Therefore, the FFT algorithms will not be considered in the subsequent discussion.

In order to do a theoretical comparison of the Gaussian integer multiplication to the real integer multiplication, three cases need to be considered:

1) The numbers are small enough to warrant the naïve multiplication method. Under this assumption, it is clear that the Gaussian integer multiplication time grows slower than

the real integer multiplication time. Assuming $t(n)$ is the time it takes to multiply two real integers of size *n* modulo *p*:

$$t_G = 3t(n) + c_G n + d_G . \tag{2.99}$$

where t_G is the time it takes to multiply two Gaussian integers of size *n* modulo *p,* c_G is the overhead (integer additions modulo *p*), and d_G is a constant overhead term. On the other hand:

$$t_r = t(2n) = 4t(n) + c_r n + d_r , \tag{2.100}$$

where t_r is the time it takes to multiply two real integers of the size 2*n* modulo *q,* c_r and d_r are the overhead terms associated with the naïve algorithm. Here c_r is smaller then c_G, while d_G and d_r terms are negligible.

$$\lim_{n\to\infty} \frac{t_G}{t_r} = \frac{3}{4} . \tag{2.101}$$

This constitutes a maximum of 25% theoretical improvement. Note that the assumption that $n \to \infty$ is incorrect because, under most implementations, at some threshold, the naïve method would be replaced by a more efficient algorithm. There is more overhead associated with the Gaussian integer multiplication, therefore, for a small *n*, t_r will be lower than t_G.

2) The numbers are sufficiently large to warrant the Karatsuba multiplication, but not large enough to warrant the Toom-3 multiplication. In this case, t_G would be represented by the same formula as in 1). Assuming $t(n)$ is the time it takes to multiply two real integers of size *n* modulo *p*:

$$t_G = 3t(n) + c_G n + d_G . \tag{2.102}$$

where t_G is the time it takes to multiply two Gaussian integers of size *n* modulo *p,* c_1 is the overhead (integer additions modulo *p*).

On the other hand, using the recursive step:

$$t_r = t(2n) = 3t(n) + c_r n + d_r . \tag{2.103}$$

where t_r is the time it takes to multiply two real integers of size 2*n* modulo *q,* c_r is the overhead associated with the Karatsuba algorithm (it is the time related to the number of additions). Here c_r is about twice the size of c_G, while d_G and d_r terms are negligible.

Since c_r is about twice as large as c_G, it is safe to conclude that, under the Karatsuba multiplication, $c_G < c_r$ and $\lim_{n\to\infty} \frac{t_G}{t_r} = 1$. There is more overhead associated with the Gaussian integer multiplication, therefore, for small *n,* t_r will be lower than t_G.

3) The numbers are sufficiently large to warrant the Toom-3 multiplication. Under this assumption, the real integer multiplication time grows slower than the Gaussian integer multiplication. Assuming $t(n)$ is the time it takes to multiply two real integers of size *n* modulo *p*:

$$t_G = 3t(n) + c_G n + d_G . \tag{2.104}$$

where t_G is the time it takes to multiply two Gaussian integers of size *n* modulo *p,* c_G is the overhead (integer additions modulo *p*), and d_G is a constant overhead term. On the other hand:

$$t_r = t(2n) = 2.76t(n) + c_r n + d_r . \tag{2.105}$$

where t_r is the time it takes to multiply two real integers of size 2*n* modulo *q,* c_r and d_r are the overhead terms associated with the naïve algorithm.

$$\lim_{n\to\infty} \frac{t_G}{t_r} \approx \frac{3}{2.76} \approx 1.087 . \tag{2.106}$$

This means that the Gaussian integer multiplication is about 9% slower under this setting. Note that the assumption that $n \to \infty$ is incorrect because, under most implementations, at some threshold, the Toom-3 method would be replaced by more efficient algorithm. There is more overhead associated with the Gaussian integer multiplication, therefore, for small n t_r will be lower than t_G.

The same arguments apply to squares, even though the squaring is generally faster than the multiplication. In theory, the squaring is up to twice as fast as multiplication, because the multiplication can be done using two squares:

$$ab = \frac{(a+b)^2 - (a-b)^2}{4}. \tag{2.107}$$

In practice, however, the square is much slower then half of a multiplication. The GMP manual [34] states that a square is around 1.5 times faster than a multiplication, if the library settings are optimized. Incidentally, the Gaussian integer squaring is also 1.5 times faster relatively to the Gaussian integer multiplication on all platforms (refer to Algorithm 1.3.1 and Algorithm 1.3.2). Therefore, the relationship between the speed of the multiplication and the square is the same for Gaussian integers.

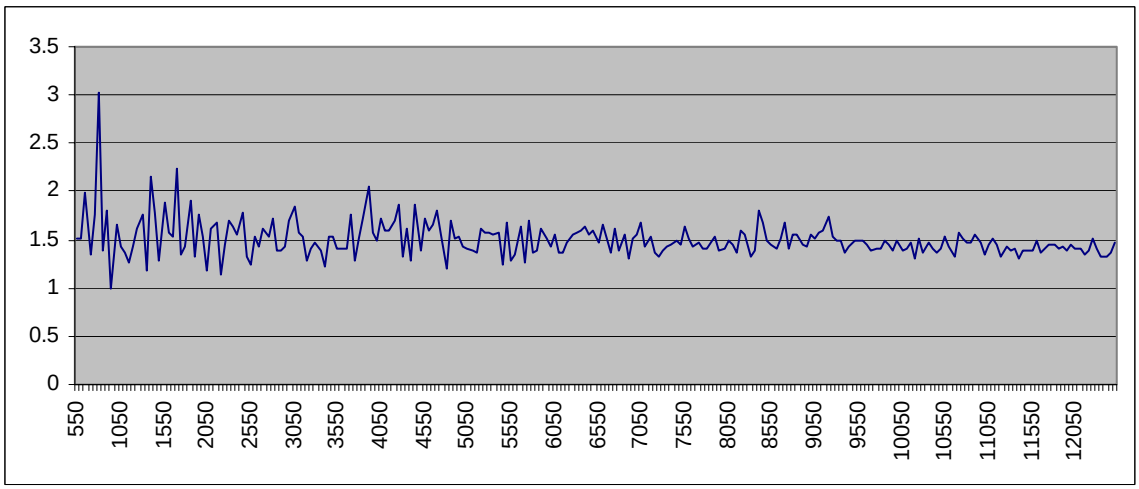

Figure 2.1 The ratio of the running time of multiplication of two numbers of the equal size vs. the running time of square of a number of the same size. The graph represents a typical performance of GMP 5.0.1 library on various platforms.

It is possible that the threshold cutoff size would lie between n and $2n$ (for example the real integer multiplication will be done with the Toom-3 algorithm, but the Gaussian integer multiplication will be done under the Karatsuba or the naïve algorithm). In this case, the exact thresholds would have to be known in order to compare the two multiplications (or squares) correctly. Unfortunately, the exact thresholds at which the one multiplication algorithm is more efficient then the other are heavily dependant on the architecture and the implementation. The bit count of thresholds varies widely among various architectures and implementations.

Figure 2.2 and Figure 2.3 show the threshold values for different squaring algorithms used by GMP 5.0.1. The values are compiled during the installation of the library. The purpose of this figure is to give a sense for the threshold values. The figures show the following:

1) the thresholds for squares tend to be higher than for multiplications
2) the thresholds tend to be larger for 64 bit CPUs than for 32 bit CPUs.

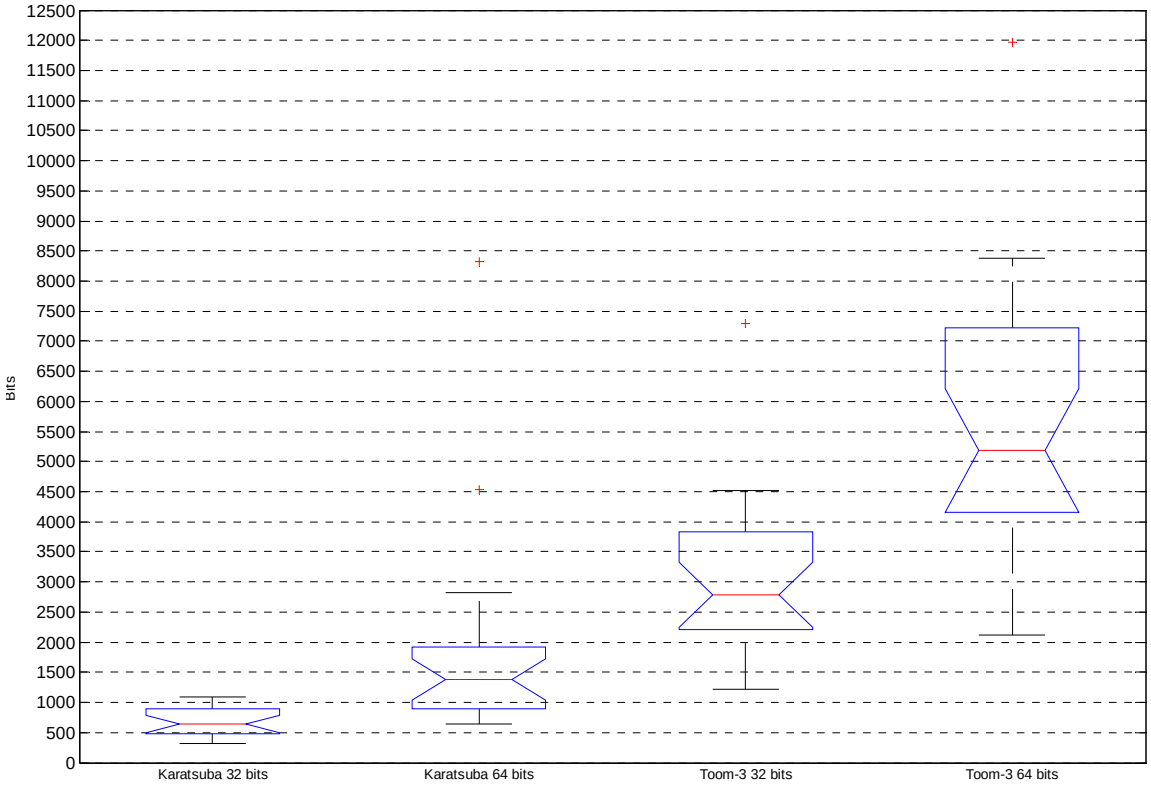

Figure 2.2 The distribution of optimal multiplication thresholds among various platforms for GMP 5.0.1.

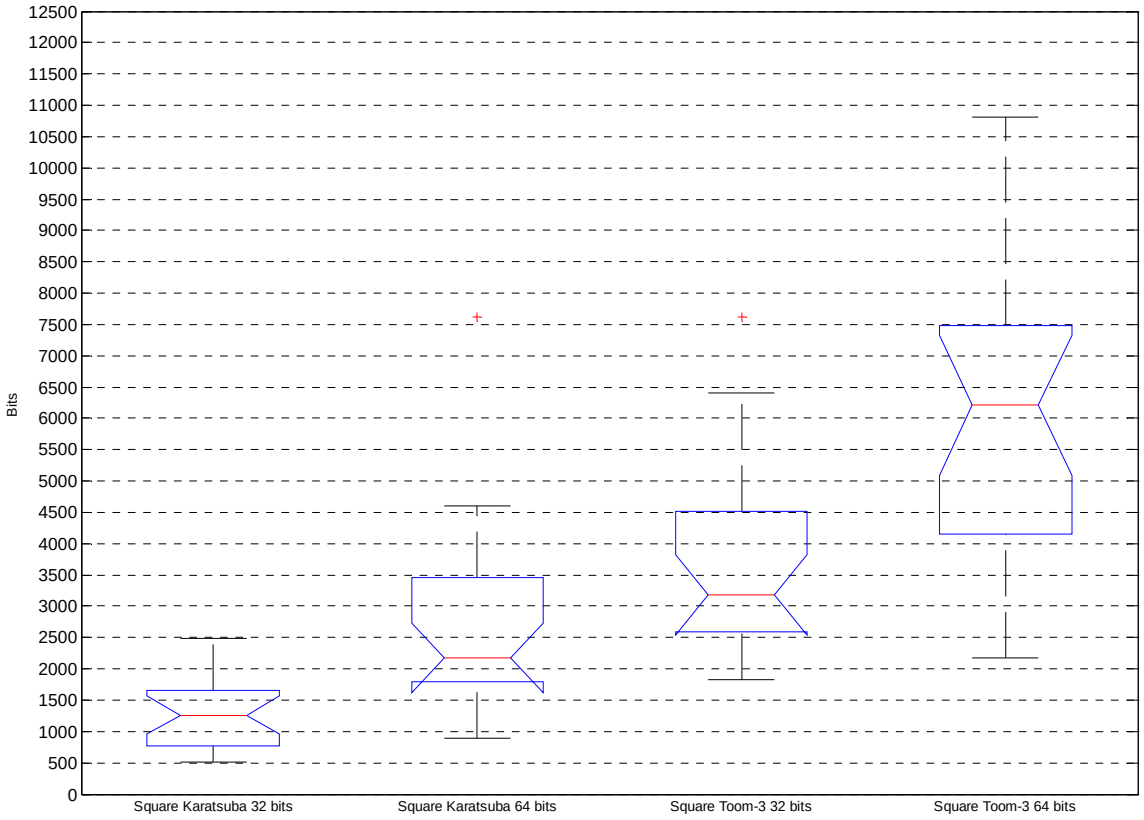

Figure 2.3 The distribution of optimal square thresholds among different platforms and counts for GMP 5.0.1.

Suppose t_G is the time it takes to multiply two Gaussian integers of size n modulo p and t_r is the time it takes to multiply two real integers of size $2n$ modulo q. Suppose that both multiplications are performed with the same multiplication algorithm that has the running time of

$$\mathrm{O}(n^{\alpha}). \tag{2.108}$$

(i.e., $\alpha = 2$ for naïve, $\alpha = 1.585$ for Karatsuba and $\alpha = 1.465$ for Toom-3 multiplication). As mentioned before, the FFT multiplication algorithms are not considered, so the formula (2.108) is sufficient to describe all applicable multiplication algorithms for the analysis. For simplicity, an assumption can be made that both multiplications are performed using the same algorithm, even though, in reality, it is possible that the threshold cutoff size would lie between *n* and 2*n* (for example the real integer multiplication will be done with the Toom-3 algorithm, but the Gaussian integer multiplication will be done with the Karatsuba or naïve algorithm). In this case, the exact thresholds have to be known to compare the multiplications (or squares) correctly. As mentioned above, the exact thresholds at which the one multiplication algorithm is more efficient then the other are heavily dependant on the architecture and the implementation. Moreover, this assumption would make the real integer exponentiation look slightly faster. Therefore, this assumption could be allowed without compromising the proof that the Gaussian integer exponentiation is faster. The overhead (lower order operations like additions or subtractions) is ignored. Under these assumptions:

$$\lim_{n\to\infty} \frac{t_G}{t_r} = \frac{3n^\alpha}{(2n)^\alpha} = \frac{3}{2^\alpha}. \tag{2.109}$$

The ratio for squares is the same, assuming the real integer square is $\frac{2}{3}t_r$:

$$\lim_{n\to\infty} \frac{s_G}{s_r} = \frac{2n^\alpha}{\frac{2}{3}(2n)^\alpha} = \frac{3}{2^\alpha}. \tag{2.110}$$

where s_G is the time it takes to multiply two Gaussian integers of size n modulo p and s_r is the time it takes to multiply two real integers of size $2n$ modulo q.

From (2.109) it is clear that $t_G < t_r$ under the naïve multiplication algorithm, $t_G \approx t_r$ under the Karatsuba multiplication and $t_G > t_r$ under the Toom-3 multiplication. As stated before, the cutoff thresholds vary widely it would be an impossible task to analyze all the possible combinations of platforms and bit counts. It is obvious with some combinations of platforms and bit counts real integer multiplication will be faster, and, with many, the Gaussian integer multiplication will be faster.

Fortunately, even though it is hard to answer definitively which multiplication is faster, it is possible to say that the Gaussian integer multiplication *modulo p* is faster. The differences between two multiplications are insignificant compared with the advantages of the "mod" operation for Gaussian integers. Therefore, the Gaussian integer exponentiation turns out to be faster.

The modulo division operation is much slower than the multiplication. For small to moderate integer sizes, the divide-and-conquer division algorithm ([16]) is commonly used. The speed of this division algorithm depends on the multiplication algorithm used. If $M(n) = Dn^c$ is the multiplication time and $T(n)$ is the division time of an integer of size $2n$ by an integer of size n, then

$$T(n) < \frac{1}{2^{c-1}-1} M(n) + \mathrm{O}(n \log n). \tag{2.111}$$

This translates to

$$T(n) < n^2 + \mathrm{O}(n \log n) \tag{2.112}$$

for the naïve multiplication,

$$T(n) < 2n^{1.585} + \mathrm{O}(n \log n) \tag{2.113}$$

for the Karatsuba multiplication (which agrees with [15]), and to

$$T(n) < 2.63n^{1.465} + \mathrm{O}(n \log n) \tag{2.114}$$

for the Toom-3 multiplication. ([16],[40],[34]). In practice, the speed of the division is about two to four times slower than the speed of the multiplication for moderately large integer sizes (section 16.2.3 of [34],[40]).

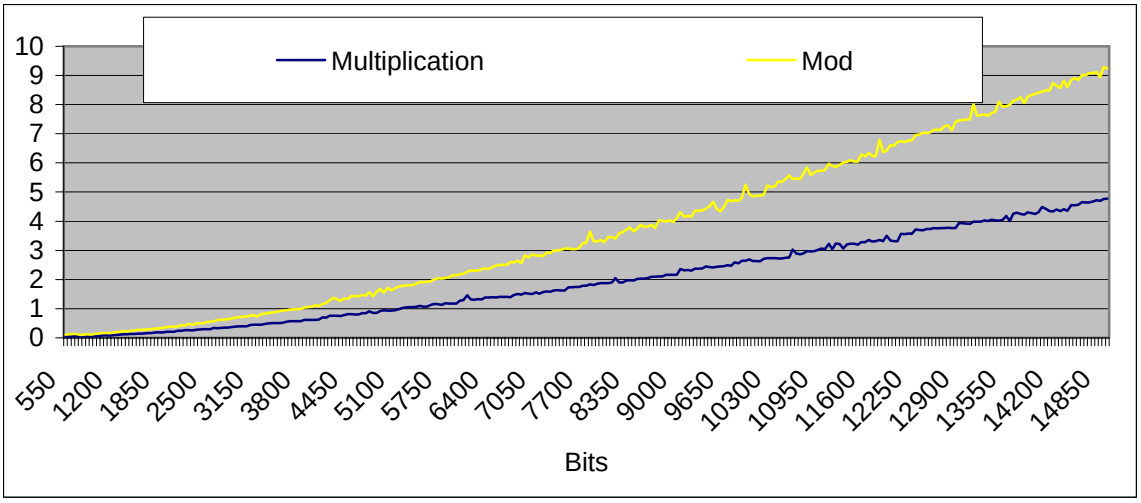

Figure 2.4 Running time of mod operation versus multiplication using GMP 5.0.1 library on AMD Opteron Model 2218 @2.6 GHz Dual core, 8GB of RAM, RHEL Linux 4.2 kernel 2.6.9 (64 bit).

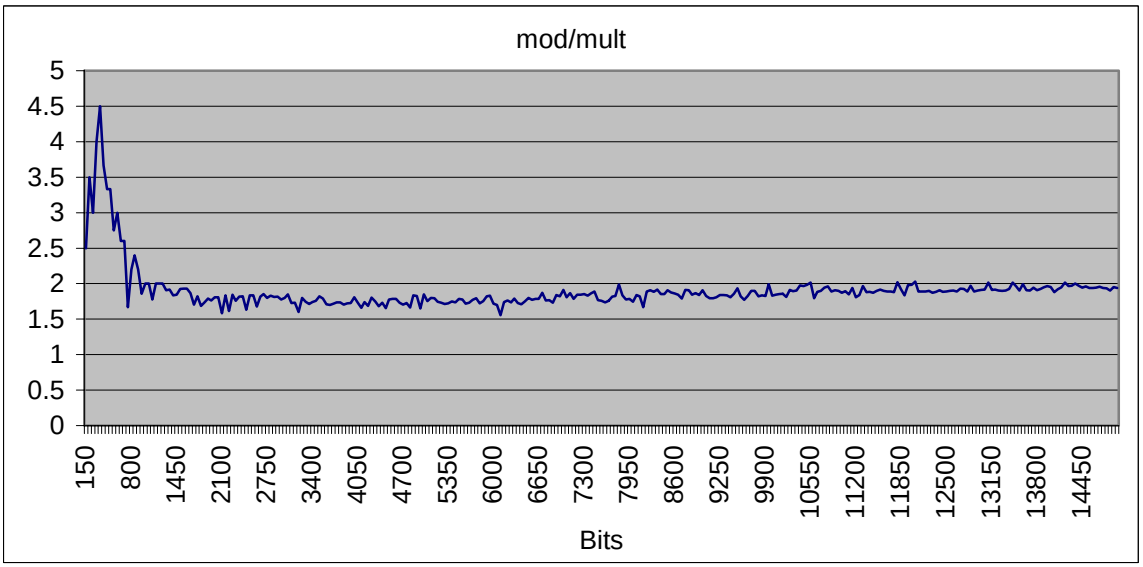

Figure 2.5 Running time of mod operation divided by the running time of multiplication using GMP 5.0.1 library on AMD Opteron Model 2218 @2.6 GHz Dual core, 8GB of RAM, RHEL Linux 4.2 kernel 2.6.9 (64 bit).

In exponentiation algorithms, the costly *mod* operation is replaced with the Montgomery reduction or *REDC*() operation ([55]), which is more efficient. It is possible to implement Montgomery reduction with the Gaussian integer exponentiation also. In fact, the Gaussian integer exponentiation still retains its speed advantages with Montgomery reduction used in place of *mod.* The *REDC*() operation speed varies depending on a platform and an implementation from about 1.2 multiplications to two multiplications ([14]). With GMP 5.0.1, the average is about 1.5 multiplications.

Whether the modulo division or the Montgomery *REDC*() function is used, the speed of the reduction at each multiplication or square step can be expressed as

$$R(n) = \beta t_r(n), \tag{2.115}$$

where $R(n)$ is the division time of an integer of size $2n$ by an integer (prime in this case) of size n, $t_r(n)$ is the multiplication time of two real integers of size n, and $\beta : 1 < \beta < 4$ is some constant.

Suppose T_G is the time required for a multiplication with a reduction for Gaussian integers and T_r is the time required for a multiplication with a reduction for real integers.

$$\lim_{n\to\infty} \frac{T_G}{T_r} = \frac{3t_r(n) + 2R(n)}{t_r(2n) + R(2n)} = \frac{3t_r(n) + 2\beta t_r(n)}{2^\alpha t_r(n) + 2^\alpha \beta t_r(n)} = \frac{3 + 2\beta}{2^\alpha (1 + \beta)}. \tag{2.116}$$

For squares, the ratio would be slightly different:

$$\lim_{n\to\infty}\frac{S_G}{S_r}=\frac{2t_r(n)+2R(n)}{\frac{2}{3}t_r(2n)+R(2n)}=\frac{6t_r(n)+6\beta t_r(n)}{2^\alpha 2t_r(n)+3*2^\alpha \beta t_r(n)}=\frac{6(1+\beta)}{2^\alpha(2+3\beta)}. \quad (2.117)$$

where S_G is the time required for a multiplication with a reduction for Gaussian integers and S_r is the time required for a multiplication with a reduction for real integers.

Table 2.5 Summarized Estimates of the Multiplication Running Time Ratio Based on the Formula (2.116)

	Naïve α=2	Karatsuba α=1.585	Toom-3 α=1.465	Toom-4 α=1.4037
β=1	0.63	0.83	0.91	0.94
β=1.2	0.61	0.82	0.89	0.93
β=1.4	0.60	0.81	0.88	0.91
β=1.5	0.60	0.80	0.87	0.91
β=1.7	0.59	0.79	0.86	0.90
β=2	0.58	0.78	0.85	0.88
β=2.2	0.58	0.77	0.84	0.87
β=2.5	0.57	0.76	0.83	0.86
β=3	0.56	0.75	0.82	0.85
β=4	0.55	0.73	0.80	0.83

Table 2.6 Summarized Estimates of the Square Running Time Ratio Based on the Formula (2.117)

	Naïve α=2	Karatsuba α=1.585	Toom-3 α=1.465	Toom-4 α=1.4037
β=1	0.60	0.80	0.87	0.91
β=1.2	0.59	0.79	0.85	0.89
β=1.4	0.58	0.77	0.84	0.88
β=1.5	0.58	0.77	0.84	0.87
β=1.7	0.57	0.76	0.83	0.86
β=2	0.56	0.75	0.82	0.85
β=2.2	0.56	0.74	0.81	0.84
β=2.5	0.55	0.74	0.80	0.84
β=3	0.55	0.73	0.79	0.82
β=4	0.54	0.71	0.78	0.81

As Table 2.5 and Table 2.6 show, the Gaussian integer exponentiation is faster on all platforms because the underlying multiplication and square operations (combined with Montgomery or modulo reductions) are faster for Gaussian integers. The exact speedup would depend on a platform, integer sizes, and the exponentiation algorithm logic (i.e., number of multiplications and squares). Realistically, for the integer sizes used for Public Key cryptography (1000-4000 bits) the expected speedup is around 20% with Gaussian integers. The rational for this is that the estimated speedup ratios for multiplications and squares for Montgomery reduction with $\beta = 1.5$ and the Karatsuba multiplication are 0.80 and 0.77, so, regardless of the ratio of multiplications/squares in a particular exponentiation algorithm, the combined ratio would be under 0.80 (i.e., 20% speedup).

2.7 Computation of Lucas Sequences

Theorem 2.5.1 and Theorem 2.5.2 show the relationship between Gaussian integers and Lucas sequences. According to these theorems, it is possible to use one to compute the other. In this section, an existing algorithm for computing Lucas sequences is reviewed and an improvement is introduced. In [74], the efficient algorithm was published to compute V_n for Q=1. Below, this algorithm is extended to compute both U_n and V_n by using the following relation:

$$U_k = \frac{2V_{k+1} - PV_k}{P^2 - 4Q}. \tag{2.118}$$

Algorithm 2.7.1 Computation of Lucas Sequences with $Q = 1$

Inputs: $k = \sum_{i=0}^{n-1} k_i 2^i$, where $n = \lceil \log_2 k \rceil$

(P, Q = 1) – Lucas sequence parameters

Outputs: (V_k, U_k)

1. $V_l := 2$; $V_h := P$;
2. **for** j=n-1 **downto** 0
3. **if** ($k[j] = 1$)
4. $V_l := V_h * V_l - P$;
5. $V_h := V_h * V_h - 2$;
6. **else**
7. $V_h := V_h * V_l - P$;

8. $V_l := V_l * V_l - 2;$

9. **endif**

10. **endfor**

11. $U_k := (2*V_h - P*V_l)/(P*P - 4*Q);$

12. **return** (V_l, U_k)

Note that the number of multiplications in Algorithm 2.8.1 is $2\lceil \log_2 k \rceil + c$, where c is a small constant. Additionally, the algorithm can be used to compute U_n and V_n modulo prime p.

The algorithm to compute both U_n and V_n for any P and Q was published in [42] . Such an algorithm could be useful for various purposes. As an example, the authors suggest using it to compute the order of an elliptic curve. It is useful for cryptosystems based on exponentiation of Gaussian integers as will be discussed in subsequent sections. The improvement to the algorithm published in [42] was published in [47]:

Algorithm 2.7.2 Computation of Lucas Sequences for any P and Q

Inputs: $k = \sum_{i=0}^{n-1} k_i 2^i$, where $n = \lceil \log_2 k \rceil$

(P, Q) – Lucas sequence parameters

Outputs: (V_k, U_k)

1. $V_l := 2$; $V_h := P$;

2. $Q_l := 1$; $Q_h := 1$;

3. **for** $j = n-1$ **downto** 0

4. $Q_l := Q_l * Q_h$;

5. **if** ($k[j] = 1$)

6. $Q_h := Q_l * Q$;

7. $V_l := V_h * V_l - P * Q_l$;

8. $V_h := V_h * V_h - 2 * Q_h$;

9. **else**

10. $Q_h := Q_l$;

11. $V_h := V_h * V_l - P * Q_l$;

12. $V_l := V_l * V_l - 2 * Q_h$;

13. **endif**

14. **endfor**

15. $U_k := (2*V_h - P*V_l)/(P*P - 4*Q)$;

16. **return** (V_l, U_k)

Note that Algorithm 2.7.2 for $Q = 1$ or $Q = -1$ still has same running time as Algorithm 2.7.1**.** The number of multiplications in Algorithm 2.7.2 is $2\lceil \log_2 k \rceil + c$, where c is a small constant.

2.8 Exponentiation of Gaussian Integers

As shown in the previous sections, the Gaussian integer exponentiation is faster then the real integer exponentiation when real integers are replaced with the Gaussian integers. However, an even faster exponentiation algorithm for Gaussian integers can be devised.

It is based on the relationship between Gaussian integers and Lucas sequences described in [50].

Algorithm 2.8.1 Lucas sequence Exponentiation of Gaussian integers (LSEG)

Inputs: (a,b) Gaussian integer

p – prime such that $p \bmod 4 = 3$

n - exponent

Output: Gaussian integer $(x,y) = (a,b)^n \bmod p$

1. $r := \left|(a,b)\right|^{\frac{p+1}{4}} \bmod p$
2. $(c,d) = r^{-1}(a,b) \bmod p$
3. $h = r^{n \bmod (p-1)} \bmod p$
4. $m = n \bmod (2(p+1))$
5. **if** $\left(r^2 == \left|(a,b)\right| (\bmod p)\right)$
6. Compute Lucas sequences $V_m(P = 2c, Q = 1)$ and $U_m(P = 2c, Q = 1)$
7. **else**
8. Compute Lucas sequences $V_m(P = 2c, Q = -1)$ and $U_m(P = 2c, Q = -1)$
9. **endif**
10. $x = hV_m \dfrac{p+1}{2} \bmod p$
11. $y = hU_m c \bmod p$
12. **return** (x,y)

Given a Gaussian integer (a,b), prime $p: p \bmod 4 = 3$ and $n: 0 < n < p^2 - 1$, suppose the aim is compute $(a,b)^n \bmod p$. First step is to compute

$$r = \left|(a,b)\right|^{\frac{p+1}{4}} \bmod p. \tag{2.119}$$

r would be the square root of $|(a,b)|$ if a square root exists. If $|(a,b)|$ is QNR (i.e., the square root does not exist) modulo p, then $r = \sqrt{-\left|(a,b)\right|} \bmod p$. Next step is to compute

$$(c,d) = r^{-1}(a,b) \bmod p. \tag{2.120}$$

It is important to note that $\left|(c,d)\right| \bmod p = 1$ or -1, because:

$$\left|(c,d)\right| = \left(\frac{a}{r}\right)^2 + \left(\frac{b}{r}\right)^2 = \frac{a^2+b^2}{r^2} (\bmod p). \tag{2.121}$$

r^2 is either $-\left|(a,b)\right| \bmod p$ or $\left|(a,b)\right| \bmod p$ depending on whether $|(a,b)|$ is QNR or not. (a,b) was factored into a product of a real integer r and Gaussian integer (c,d), and $\left|(c,d)\right| \bmod p = 1$ or -1:

$$(a,b) = r(c,d) \bmod p. \tag{2.122}$$

To compute $(a,b)^n \bmod p$ the following values have to be computed:

$$r^{n \bmod (p-1)} \bmod p \tag{2.123}$$

and

$$(c,d)^n \bmod p\,. \tag{2.124}$$

To compute (2.123) it is possible to use any available real integer exponentiation modulo prime algorithm (the order of real integer modulo p is $p-1$, so n can be reduced modulo $p-1$). In order to compute (2.124), the relationship between Gaussian integers and Lucas sequences described in algorithm in [50] could be used. To compute (2.124), it is enough to compute $V_m\left(2c,\left|(c,d)\right|\right)$, $U_m\left(2c,\left|(c,d)\right|\right)$ and

$$(c,d)^n = (c,d)^m = \left(\frac{V_m}{2}, U_m d\right) (\bmod p)\,, \tag{2.125}$$

where $m = n \bmod (2(p+1))$ (the order of Lucas sequences with $Q=1$ is $p+1$ (Lemma 2.3.3) and with $Q=-1$ is $2(p+1)$ (Lemma 2.3.5)). V_m and U_m could be efficiently computed using any published algorithm to compute Lucas sequences, such as [74] or [68]. The algorithms published in [74] and [68] would only compute $V_m(2c,1)$, however, they can be easily enhanced to compute $V_m(2c,-1)$, $U_m(2c,-1)$ or $U_m(2c,1)$. Finally few more multiplications are needed to get the final answer:

$$(a,b)^n = r^n (c,n)^n \pmod{p}. \tag{2.126}$$

Example 2.8.1 Gaussian integer exponentiation with LSEG Algorithm

Inputs: $(a,b) = (2,5)$ Gaussian integer

$p = 23$ – prime

$n = 423$ – exponent

Output: Gaussian integer $(x,y) = (a,b)^n \bmod p$

1. $r := \left|(a,b)\right|^{\frac{p+1}{4}} \bmod p = \left(2^2 + 5^2\right)^{\frac{23+1}{4}} \bmod 23 = 6^6 \bmod 23 = 12$
2. $(c,d) := r^{-1}(a,b) \bmod p = 12^{-1}(2,5) \bmod 23 = 2(2,5) \bmod 23 = (4,10)$
3. $h := r^{n \bmod (p-1)} \bmod p = 12^{423 \bmod 22} \bmod 23 = 12^5 \bmod 23 = 18$
4. $m := n \bmod (2(p+1)) = 423 \bmod 48 = 39$
5. $\left(r^2 == \left|(a,b)\right| (\bmod p)\right)$ is true: $12^2 \bmod 23 == 6$, therefore:
6. Compute $V_m(P = 2c, Q = 1) = V_{39}(8,1) = 18 (\bmod 23)$
7. Compute $U_m(P = 2c, Q = 1) = U_{39}(8,1) = 6 (\bmod 23)$
8. $x = hV_m \frac{p+1}{2} \bmod p = 18 * 18 * 12 \bmod 23 = 1$
9. $y = hU_m c \bmod p = 18 * 6 * 10 \bmod 23 = 22$
10. **return** $(x,y) = (1,22)$

The speed advantage of Algorithm 2.8.1 is due to the fact that it does less work. Note that the most expensive operations in this algorithm are two real integer exponentiations (lines 1 and 3) and one Lucas sequences computation (line 6 or line 8).

The speed advantage of Algorithm 2.8.1 would vary depending on the exponentiation algorithm used and there are too many variations to consider. To illustrate the speed advantage of Algorithm 2.8.1, it can be compared to the sliding window Montgomery Gaussian integer exponentiation algorithm, same as the one implemented by GMP ([34] section 16.4.2, [54] algorithm 14.85). The exponentiation algorithm modulo prime implemented by GMP is a highly efficient implementation and is widely used for cryptographic algorithms. Moreover, the sliding window exponentiation algorithm holds advantage over many other exponentiation algorithms for average case and random exponent ([45], [33]) and, therefore, it is used by GMP library for modular exponentiation.

The sliding window Montgomery reduction exponentiation algorithm for Gaussian integers is denoted as SWG (Sliding Window Gaussian). It is the same algorithm as the one implemented by GMP, but with real integers replaced with Gaussian integers modulo prime $p : p \bmod 4 = 3$. The prime p is n bits long. Suppose the window size is w and the exponent is $e : 0 < e < p^2 - 1$. Suppose also that e is such that the number of multiplications is $\frac{2n}{w}$ (the best case for sliding window algorithm). This assumption is reasonable because mostly random looking exponents are used for cryptographic applications. In case this assumption is not true, the SWG algorithm would be even slower compared to Algorithm 2.8.1**.** Suppose $t_r(n)$ is the running time of one multiplication of two integers of n bits long and T_{SWG} is the running time of SWG algorithm (ignoring lower order operations like additions). For each Gaussian integer multiplication, three real integer multiplications $t_r(n)$ and two Montgomery reduction

operations *REDC*() have to be performed. For each Gaussian integer square two real integer multiplications $t_r(n)$ and two Montgomery reduction operations *REDC*() have to be performed. Each *REDC*() operation has a cost of $\beta t_r(n)$. The precomputation required for the sliding window algorithm could be ignored, it is larger for SWG algorithm, but becomes less significant as *n* grows. Thus:

$$T_{SWG} = 2n\left(2t_r(n) + 2\beta t_r(n)\right) + \frac{2n}{w}\left(3t_r(n) + 2\beta t_r(n)\right) = \\ = \left(2 + 2\beta + \frac{3+2\beta}{w}\right)2nt_r(n) \quad . \tag{2.127}$$

For Algorithm 2.8.1 the same sliding window exponentiation with Montgomery reduction for real integer exponentiation can be used. Two real integer exponentiations with exponents less than half the size of *e* in bits have to be done. To compute Lucas sequences, the algorithm [74] could be used. The size of the exponent for this algorithm is approximately one half of the size of *e* (i.e., *n*, not 2*n*) and for each bit one multiplication, one square and two *REDC*() operations are required. It can be assumed that the square takes $\frac{2}{3}$ of the time of multiplication. Suppose T_{LSEG} is the running time of Algorithm 2.8.1**,** ignoring lower order operations, like additions. As with SWG, precomputation required for sliding window algorithm could be ignored, noting that it is smaller for Algorithm 2.8.1**.** Finally:

$$T_{LSEG} = 2\left(n\left(\frac{2}{3}t_r(n) + \beta t_r(n)\right) + \frac{n}{w}\left(t_r(n) + \beta t_r(n)\right)\right) +$$
$$+n\left(\frac{2}{3}t_r(n) + \beta t_r(n) + t_r(n) + \beta t_r(n)\right) = \quad . \qquad (2.128)$$
$$= 2nt_r(n)\left(1.5 + 2\beta + \frac{1+\beta}{w}\right)$$

Thus the improvement based on various window sizes and values of β could be estimated.

Table 2.7 T_{LSEG} / T_{SWG} Ratio for Various β and Window Sizes

	w=1	w=2	w=3	w=4	w=5	w=6	w=7	w=8
β=1	0.61	0.69	0.74	0.76	0.78	0.79	0.80	0.81
β=1.2	0.62	0.70	0.75	0.77	0.79	0.81	0.81	0.82
β=1.4	0.63	0.71	0.76	0.78	0.80	0.82	0.82	0.83
β=1.5	0.64	0.72	0.76	0.79	0.81	0.82	0.83	0.84
β=1.7	0.64	0.73	0.77	0.80	0.81	0.83	0.84	0.84
β=2	0.65	0.74	0.78	0.81	0.82	0.84	0.85	0.85
β=2.2	0.66	0.74	0.79	0.81	0.83	0.84	0.85	0.86
β=2.5	0.67	0.75	0.79	0.82	0.84	0.85	0.86	0.87
β=3	0.68	0.76	0.80	0.83	0.85	0.86	0.87	0.88
β=4	0.69	0.77	0.82	0.84	0.86	0.87	0.88	0.89

As illustrated by Table 2.7, Algorithm 2.8.1 offers an improvement approximately 18% over SWG for the window size relevant to real world cryptography applications (1000-4000 bits and window size 7). This translates to about 35% theoretic improvement over

SWR, but in practice it would be much higher because of low overhead associated with Algorithm 2.8.1. Moreover, the real time could be significantly improved if (2.123) and (2.124) are computed in parallel.

It is easy to estimate the real running time improvement ratio of Algorithm 2.8.1 (LSEG) implemented with threads that compute (2.123) and (2.124) in parallel. Such algorithm shell be denoted as LSEG*. It is easy to see that the computation of (2.123) will be faster than the computation of (2.124) (real integer exponentiation is faster than Lucas sequence computation for the same prime). Therefore, the running time of LSEG* as follows can be estimated as follows:

$$\begin{aligned} T_{LSEG*} &= n\left(\frac{2}{3}t_r(n)+\beta t_r(n)\right)+\frac{n}{w}\left(t_r(n)+\beta t_r(n)\right)+ \\ &+n\left(\frac{2}{3}t_r(n)+\beta t_r(n)+t_r(n)+\beta t_r(n)\right)= \\ &= 2nt_r(n)\left(1.16666+1.5\beta+\frac{1+\beta}{2w}\right) \end{aligned} \quad . \qquad (2.129)$$

Thus the improvement based on various window sizes and values of β can be estimated.

Table 2.8 T_{LSEG^*} / T_{SWG} Ratio for Various β and Window Sizes

	w=1	w=2	w=3	w=4	w=5	w=6	w=7	w=8
β=1	0.41	0.49	0.53	0.56	0.57	0.59	0.60	0.60
β=1.2	0.41	0.50	0.54	0.56	0.58	0.59	0.60	0.61
β=1.4	0.42	0.50	0.54	0.57	0.59	0.60	0.61	0.62
β=1.5	0.42	0.51	0.55	0.57	0.59	0.60	0.61	0.62
β=1.7	0.43	0.51	0.55	0.58	0.60	0.61	0.62	0.63
β=2	0.44	0.52	0.56	0.59	0.60	0.62	0.63	0.63
β=2.2	0.44	0.52	0.56	0.59	0.61	0.62	0.63	0.64
β=2.5	0.44	0.53	0.57	0.59	0.61	0.62	0.63	0.64
β=3	0.45	0.53	0.58	0.60	0.62	0.63	0.64	0.65
β=4	0.46	0.54	0.59	0.61	0.63	0.64	0.65	0.66

As Table 2.8 shows, the multithreaded version of Algorithm 2.8.1 (LSEG***)** offers an improvement approximately 39% over SWG for the window size relevant to real world cryptography applications (1000-4000 bits and window size 7). This translates to about 52% theoretic improvement over SWR, but in practice it would be much somewhat lower because of the overhead associated with multithreaded programming. Nevertheless, it is a great improvement ratio considering this algorithm does not require any special hardware or software and can be easily implemented.

Most contemporary platforms have multiple processors and/or multiple cores. On such platforms the parallel implementation of Algorithm 2.8.1 (LSEG***)** is more advantageous because with a relatively small added cost associated with multithreading, a significant improvement in real running time was achieved. The overhead varies widely among platforms and implementations but it tends to be relatively small compared to an added benefit in real running time.

2.9 Experimental Results

For the experiments the latest release of GMP library 5.0.1 was used. On each platform, the library was installed and the optimization step performed. The language used was C compiled it with gcc compiler. The version of gcc did vary across the platforms, however, it is not important for this study, because only relative running times on the same platform compiled with the same optimization level were of interest. The sliding window exponentiation with the Montgomery reduction algorithm for Gaussian integers with the optimal sliding window size (SWG) was implemented. The optimal sliding window size was calculated for every exponent.

For real integer exponentiation, mpz_powm function was used that came with GMP library. It was implemented using the Sliding Window exponentiation algorithm (algorithm 14.85 in [54]) using Montgomery reduction (section 16.4.2 in [34]). This algorithm for real integers shall be labeled as SWR (Sliding Window Real). The SWG was implemented as efficiently as possible; however, it still has more overhead than GMP's "mpz_powm" function. Some of this overhead is due to the fact that Gaussian integer multiplications and squares require extra additions, which was ignored in the estimates. Some of this overhead is due to the fact that GMP implementations tend to be very efficient because they use low level platform specific techniques to minimize the overhead and speedup the calculations. Nevertheless, this implementation of SWG overtook SWR for bit sizes above 1000 on all platforms and showed the predicted in previous section 20% speed advantage.

Two versions of Algorithm 2.8.1 were implemented. Both versions used "mpz_powm" function for real integer exponentiation and the algorithm published in [74]

for Lucas sequences computations. The first version (LSEG) executed the steps of Algorithm 2.8.1 sequentially. For small bit sizes (500-2500 bits) it performed better than predicted, relative to the implementation of SWG. This is due to the fact that it has much less overhead than SWG, and the estimates in Table 2.5 are biased towards SWG (make SWG look faster). For really high bit sizes (>4000) the experimental results confirm the estimates in Table 2.5 (show 15% speedup), but for the bit sizes that are practical for cryptography the results show improvement of 40-20%, which is much better than predicted 18%.

The second version of Algorithm 2.8.1 (LSEG*) was implemented using threads to process real integer exponentiation and Lucas sequences computation in parallel. As Figure 2.6 demonstrates, the CPU time needed for LSEG* is slightly higher than the CPU time needed for LSEG. However, the difference is very small. It is due to the overhead associated with threads. On the other hand, the reduction in real time is very significant as Figure 2.10 demonstrates. The achieved improvements are in line with the estimates in Table 2.8. Similar results were achieved on all platforms.

The experiments were performed for bit sizes varying from 100 to 11500 bits. For each bit size, a random prime $p : p \bmod 4 = 3$ of bit size n and prime q of bit size from $2n$-1 to $2n$ were generated. The CPU performance varied widely among the platforms, so the number of trials N was calibrated for each platform, so one could differentiate between the performance of the algorithm for lower bit sizes and it would finish in a reasonable amount of time for high bit sizes. For each bit size the following numbers were randomly generated:

1) N Gaussian integers $(a,b) : 0 < a,b < p-1$

2) N exponents $e : 0 < e < p^2 - 1$

3) N real integers $c : 0 < c < q - 1$

For each of the N Gaussian integers $(a,b)^e \bmod p$ were computed using SWG, LSEG and LSEG*. Additionally, for each of the N integers c, $c^e \bmod q$ was computed using SWR. For each algorithm, the total CPU time was recorded.

The platforms used vary widely in architecture and computing power. Below is the list of them:

1) Lenovo T400 Laptop, Intel Core2 Duo CPU T9400 @2.53GHz with 3GB of RAM, Cygwin under Windows XP OS (32 bit).
2) AMD Opteron Model 2218 @2.6 GHz Dual core, 8GB of RAM, RHEL Linux 4.2 kernel 2.6.9 (64 bit).
3) SunOS 5.10, sun4u, Ultra-4, two UltraSPARC-II @ 296MHz processors, 1 GB of RAM (64 bit)

As expected, the nominal running time varied widely among platforms, but the relative running times among the algorithms remained consistent. Below are the graphs of the results from the platform 2). For the sake of brevity the graphs for other platforms weren't included, because they are very similar.

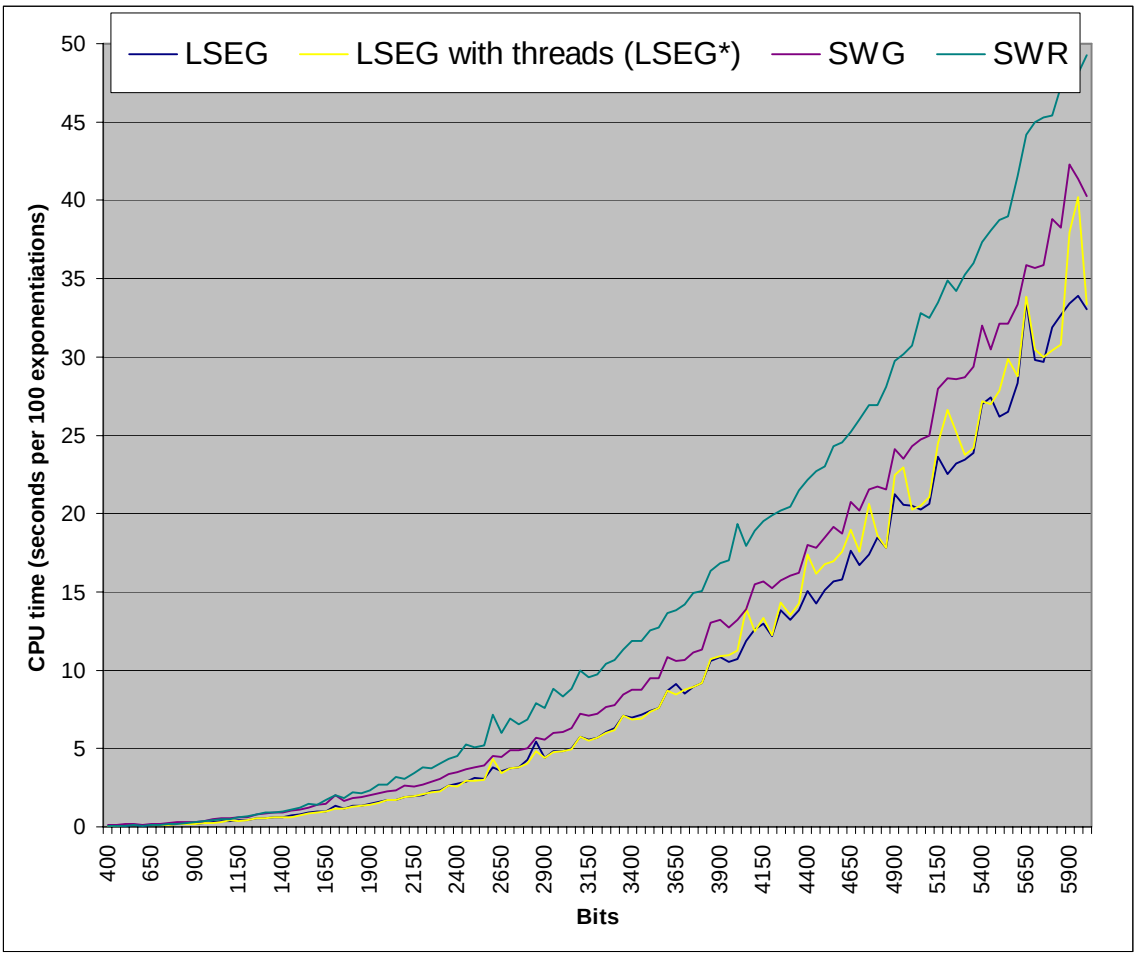

Figure 2.6 The CPU time of SWR,SWG, LSEG and LSEG* for various bit sizes.

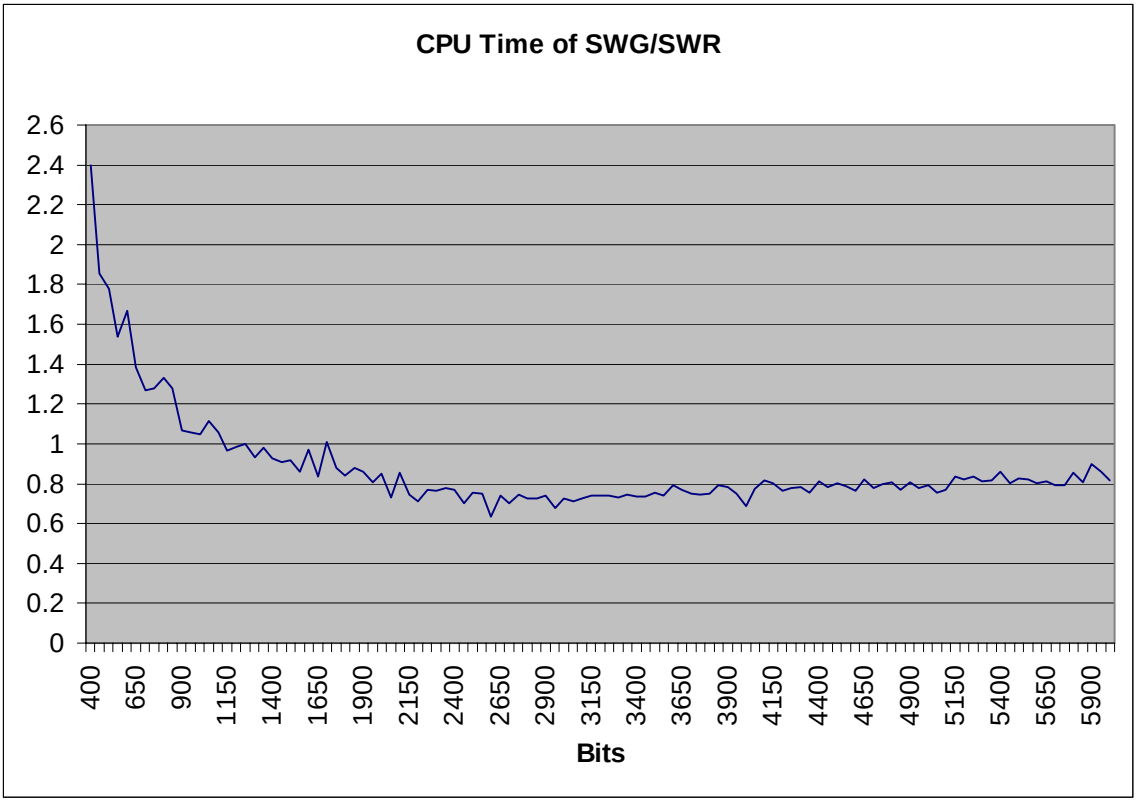

Figure 2.7 The ratio of the running time of SWG algorithm over SWR.

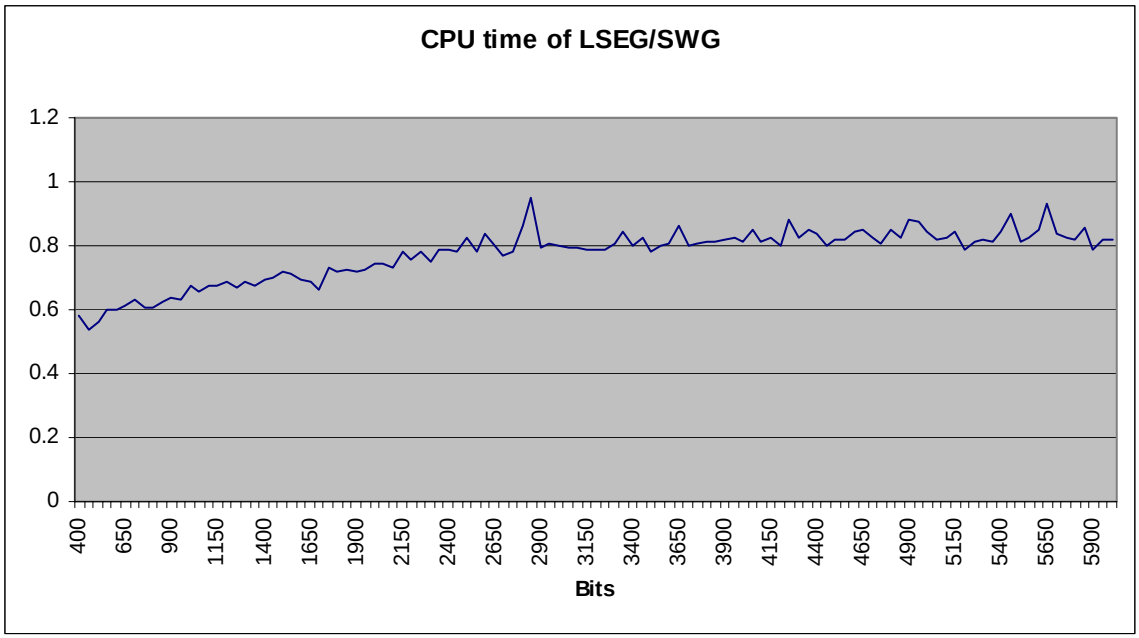

Figure 2.8 The ratio of the CPU time of Algorithm 2.8.1 (LSEG) over SWG.

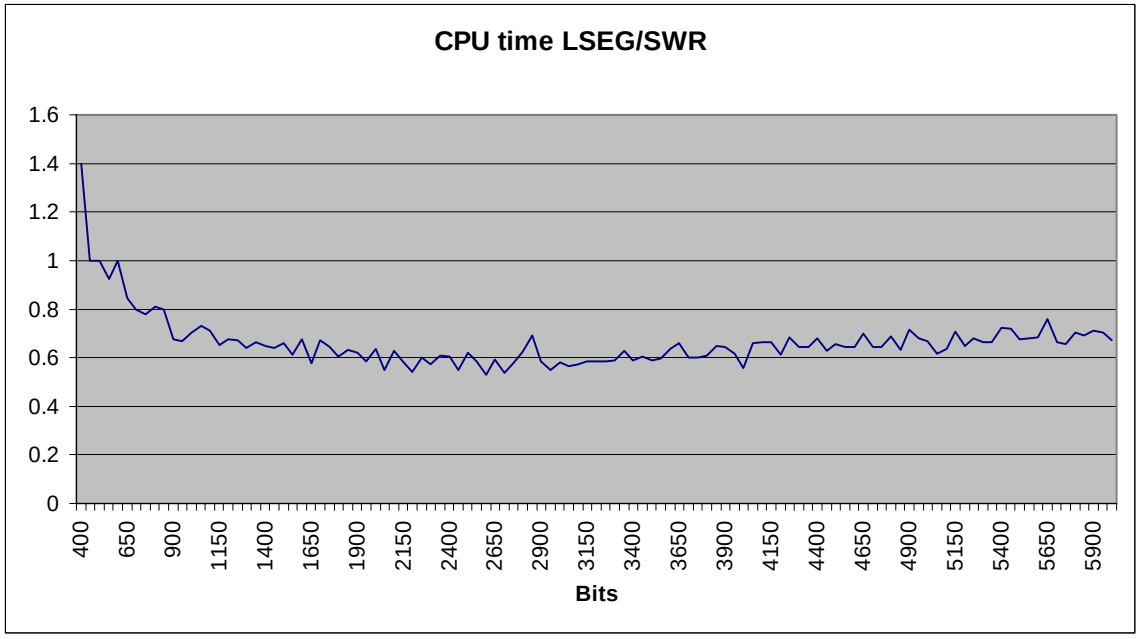

Figure 2.9 The ratio of the running time of Algorithm 2.8.1 algorithm over SWR.

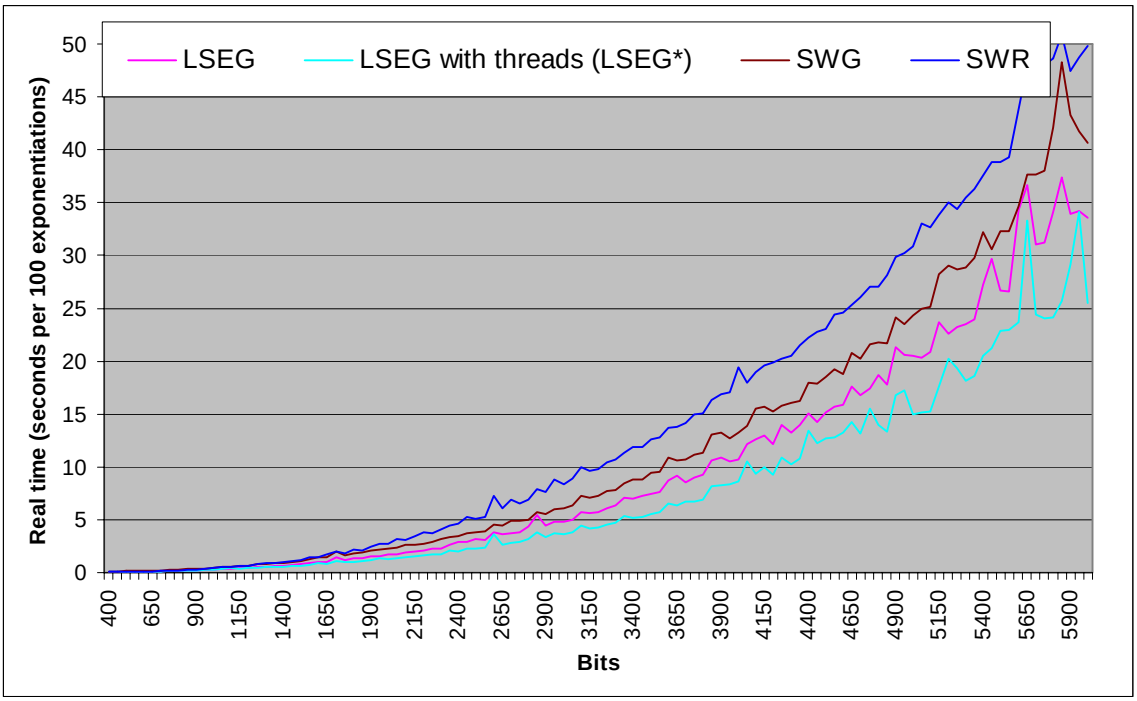

Figure 2.10 The real running time of SWR, SWG, LSEG and LSEG* for various bit sizes.

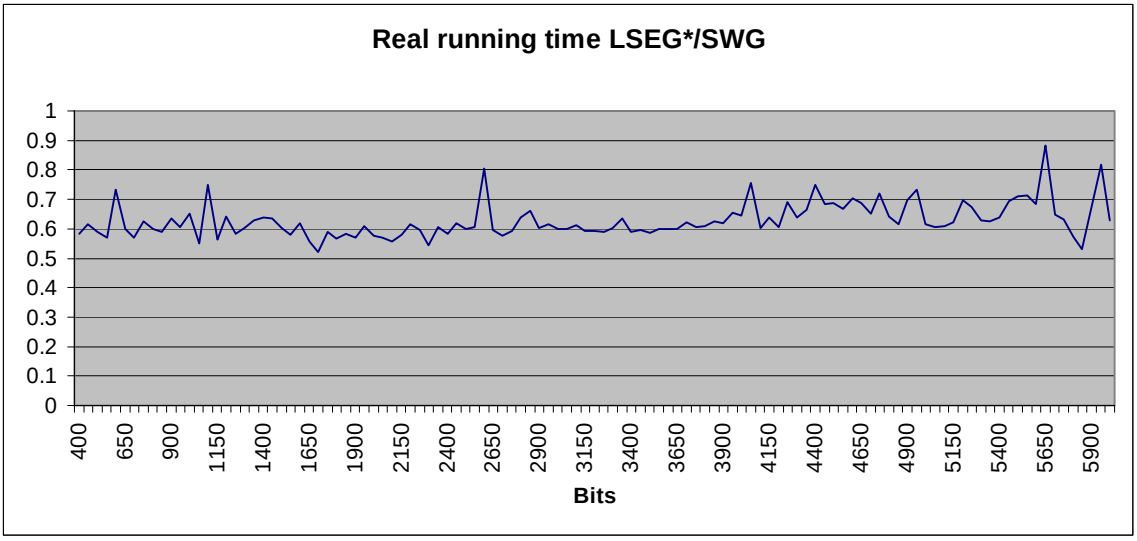

Figure 2.11 Ratio of real running time of LSEG* over SWG.

The experimental results confirm the estimates from Table 2.5, Table 2.6 , Table 2.7 and Table 2.8.

2.10 Algorithms for Finding Gaussian Generators

Algorithm 2.2.3 and Algorithm 2.2.4 necessitate a way to find Gaussian integer generators. A common algorithm based on general cyclic group properties is described below:

Algorithm 2.10.1 Simple Algorithm for finding Gaussian Generators

1. Factor p^2-1:

$$p^2 - 1 = (f_1)^{e_1}(f_2)^{e_2}...(f_k)^{e_k} \tag{2.130}$$

2. Select a G=(a,b) such that $a > 0$, $b > 0$ and $a^2 \neq b^2 \pmod{p}$

3. For each factor f_i of p^2-1, compute

$$B_i = G^{\frac{p^2-1}{f_i}} \bmod p \tag{2.131}$$

If any of B_i=(1,0) mod p then G is not a generator, then go to Step 2. Otherwise, G is a generator.

Algorithm 2.10.1 is a straightforward extension of the most common algorithm for finding real integer generators and is based on Lagrange's Theorem in the mathematics of group theory. It tests all the divisors of the largest period of a candidate to determine if it is a generator. Using the theoretical framework presented in Sections 2.3 and 2.5, it was possible to design an improved algorithm to find Gaussian integer generators for a given prime.

Algorithm 2.10.2 Norm Algorithm for finding Gaussian Generators

1. Factor $\frac{p-1}{2}$ into k_1 factors:

$$\frac{p-1}{2} = (f_1)^{e_1}(f_2)^{e_2}...(f_{k_1})^{e_{k_1}} \tag{2.132}$$

2. Factor $2(p+1)$ into k_2 factors::

$$2(p+1) = (h_1)^{e_1}(h_2)^{e_2}...(h_{k_2})^{e_{k_2}} \tag{2.133}$$

3. Select a G=(a,b) such that $a > 0$, $b > 0$ and $a^2 \neq b^2 \pmod{p}$

4. $r := \left|(a,b)\right|^{\frac{p+1}{4}} \bmod p$

5. If $r^2 = |(a,b)| \bmod p$ go to Step 4.

6. For each factor f_i of $\frac{p-1}{2}$ compute

$$b_i = \left|(a,b)\right|^{\frac{p-1}{f_i}} \bmod p \tag{2.134}$$

If any of $b_i = 1 \bmod p$ then G is not a generator, go to Step 3.

7. $(c,d) = \left(r^{-1}(a,b)\right)^2 \bmod p$

8. For each factor h_i of $2(p+1)$ compute Lucas sequence values $V_{h_i}(P = 2c, Q = 1)$.

If any of $V_{h_i} = 2 \bmod p$ then G is not a generator, go to Step 3. Otherwise, G is a generator.

The relative efficiency of the algorithms would vary somewhat with the type of exponentiation algorithm used. However, it is clear that Algorithm 2.10.2 is much more efficient than Algorithm 2.10.1, as shown below.

Algorithm 2.10.1 performs the exponentiation of Gaussian integers. This means that it is possible to use similar time complexity analysis that was done is Section 2.8 for the SWG (Sliding Window Gaussian) algorithm. There are differences, however. In the analysis for T_{SWG} (2.127), it was reasonable to assume that every exponent would be of average bit size 2*n,* where *n* is the bit size of prime *p.* For Algorithm 2.10.1, on the other hand, the exponent sizes would be different.

Suppose size(*x*) stands for the size of integer *x* in bits (e.g., size(*p*)=*n*), then the sum of sizes of the factors of p^2-1 from (2.130) is

$$\sum_{i=1}^{k} \text{size}\left(f_i\right) = \text{size}\left(p^2 - 1\right) = 2n \tag{2.135}$$

The size of each exponent of f_i used in (2.131) is

$$\text{size}\left(\frac{p^2 - 1}{f_i}\right) = 2n - \text{size}\left(f_i\right). \tag{2.136}$$

The total number of multiplications and squares performed by Algorithm 2.10.1 is

$$T_{\text{Algorithm 2.3.1}} = \left(\sum_{i=1}^{k}\left(2n - size(f_i)\right)\right)\left(2 + 2\beta + \frac{3+2\beta}{w}\right)t_r(n) =$$
$$= \left(2kn - 2n\right)\left(2 + 2\beta + \frac{3+2\beta}{w}\right)t_r(n) = \quad , \qquad (2.137)$$
$$= \left(k-1\right)\left(2 + 2\beta + \frac{3+2\beta}{w}\right)2nt_r(n)$$

where n = size(p), $t_r(n)$ - time to multiply two real integer of size n, k – number of prime factors of $p^2 - 1$ as in (2.130), w – sliding window size, and the cost of the modular reduction is $\beta t_r(n)$.

The running time complexity analysis for of Algorithm 2.10.2 is similar to the time complexity analysis of LSEG. As with Algorithm 2.10.1, the size of the exponent is different. In contrast with Algorithm 2.10.1, Algorithm 2.10.2 factors $p^2 - 1$ in parts. It factors $\frac{p-1}{2}$ in Step 2 and $2\left(p+1\right)$ in Step 3. It is worth noting that

$$\sum_{i=1}^{k_1}\text{size}\left(f_i\right) + \sum_{i=1}^{k_2}\text{size}\left(h_i\right) =$$
$$= \text{size}\left(\frac{p-1}{2}\right) + \text{size}\left(2\left(p+1\right)\right) = \qquad (2.138)$$
$$= n + n = \text{size}\left(p^2 - 1\right) = 2n$$

The purpose of this discussion is to show that Algorithm 2.10.2 is faster than Algorithm 2.10.1. Because real integer exponentiation is much faster than Lucas sequences computation, it is permissible to presume, for simplicity, that real integer exponentiation has the same time complexity as Lucas sequences computation. Subsequently,

$$
\begin{aligned}
&T_{\text{Algorithm 2.3.2}} < n\left(\frac{2}{3}t_r(n)+t_r(n)+2\beta t_r(n)\right)+\\
&+\left(\sum_{i=1}^{k_1}\left(n-size(f_i)\right)\right)\left(\frac{2}{3}t_r(n)+t_r(n)+2\beta t_r(n)\right)+\\
&+\left(\sum_{i=1}^{k_2}\left(n-size(h_i)\right)\right)\left(\frac{2}{3}t_r(n)+t_r(n)+2\beta t_r(n)\right)=\\
&=nt_r(n)\left(\frac{5}{3}+2\beta\right)+\\
&+n\left(k_1-1\right)t_r(n)\left(\frac{5}{3}+2\beta\right)+\\
&+n\left(k_2-1\right)t_r(n)\left(\frac{5}{3}+2\beta\right)=\\
&=nt_r(n)\left(\frac{5}{3}+2\beta\right)\left(k-1\right)
\end{aligned}
\quad , \tag{2.139}
$$

From this inequality, it follows that:

$$
\frac{T_{\text{Algorithm 2.3.2}}}{T_{\text{Algorithm 2.3.1}}} < \frac{\frac{5}{3}+2\beta}{2\left(2+2\beta+\frac{3+2\beta}{w}\right)} \tag{2.140}
$$

From (2.140) it is clear that Algorithm 2.10.2 is always more than twice as fast as Algorithm 2.10.1, regardless of w or β (both w and β have to be greater than 0, of course). In reality though, Algorithm 2.10.2 is even faster, because the real integer exponentiation is much faster than the Lucas sequences computation for the same prime. (For simplicity, it was assumed that the real integer exponentiation had the same speed as

the Lucas sequences computation. This made the estimate for the speed of Algorithm 2.10.2 appear higher.)

To find a real generator for p, p-1 has to be factored. To find a Gaussian Generator, both p-1 and p+1 have to be factored. One could argue that for large p it may be too hard to factor p-1 and p+1. Fortunately, for discrete logarithm based algorithms, very large prime factor of p-1 and p+1 are desired in order to protect against various attacks. If both p-1 and p+1 have large prime factors (close to the the bitsize of prime p), then factoring is easy.

2.11 Chapter Summary

In this chapter it was shown that there are no benefits to using non-Blum Gaussian primes in DLP-based Public Key (PK) cryptography algorithms because there is one-to-one relationship between Gaussian integers modulo non-Blum Gaussian primes and real integers. Consequently, the Gaussian integers considered for PK cryptosystems should be limited to primes $P = (p, 0)$, where p is a prime and $p \bmod 4 = 3$. This restriction allows for efficient implementation of MOD operation used for PK cryptosystems.

In Chapter 2, the properties of the Gaussian integer exponentiation are analyzed in-depth. Based on these properties, an improved algorithm to find a Gaussian integer generator is described (Algorithm 2.10.2, Norm Algorithm for finding Gaussian Generators). The speed of Algorithm 2.10.2 was compared to the speed of Algorithm 2.10.1 (Simple Algorithm for finding Gaussian Generators). It was proven that Algorithm 2.10.2 is always faster than Algorithm 2.10.1. Additionally, it was demonstrated that the

discrete logarithm for Gaussian integers can be computed by decomposing the Gaussian integer group into two subgroups and computing the discrete logarithm in each subgroup.

In Section 2.5, it was proven that the Gaussian integer DLP is equivalent to a combination of the Lucas sequences DLP and the real integer DLP. This fact means that the Gaussian integer DLP is harder than the real integer DLP, consequently, the PK cryptosystems based on Gaussian integers exponentiation modulo prime p that is n bits long is equivalent in security to a real integer PK DLP based cryptosystems modulo prime q which is $2n$ bits long.

Finally, based on the proof of the security of the Gaussian integer DLP in Section 2.5, the exponentiation of Gaussian integers modulo prime p were compared to the exponentiation of real integers modulo prime q, where q is twice the size of p. Firstly, it was shown (both theoretically and experimentally) that under such settings the multiplication of Gaussian integers modulo p is about 20% for the bit range currently used for PK cryptosystems (1500+ bits). Secondly, a novel exponentiation algorithm for Gaussian integers was introduced in Section 2.8: Algorithm 2.8.1, Lucas sequence Exponentiation of Gaussian integers (LSEG). It improves the speed by additional 18% (on top of 20% which results in about 34% over real integer exponentiation). Moreover, some steps of the LSEG algorithm could be run in parallel (such version of LSEG algorithm was denoted as LSEG*). LSEG* offers about 52% theoretical improvement over real integer exponentiation.

Section 2.9 describes the experiments performed on various computing platforms to validate the theoretical results described in Section 2.8. All the theoretical results were confirmed experimentally.

CHAPTER 3

EXTENSION OF RABIN CRYPTOSYSTEM INTO THE FIELD OF GAUSSIAN INTEGERS

3.1 Restriction of Gaussian Integer Domain

To extend the Rabin algorithm, a subset of $Z[i]$, as described in [27], is considered, namely, real primes *p: p* mod 4=3 or real Blum primes. This allows for the use of modulo (mod) operation as defined in Definition 1.3.2. The overhead of this mod operation is minimal.

The rationale for the restriction of the domain is that the use of Gaussian primes $P = (a,b) :| P | \bmod 4 = 1$ (or non-Blum Gaussian primes) leads to a less secure and inefficient algorithm. This point is discussed in-depth in Section 2.1.

3.2 Rabin Cryptosystem

Rabin Cryptosystem was proposed in 1979 by Michael O. Rabin. The high level description of the algorithm is below:

Algorithm 3.2.1 Original Rabin Cryptosystem

Key generation

Alice selects two distinct primes *p* and *q* and calculate *n=pq*. She publishes *n* as a public key.

Encryption

Given message m: $0 \le m \le n - 1$, Bob computes the ciphertext

$$c = m^2 \bmod n . \tag{3.1}$$

Decryption

Given ciphertext c Alice computes square roots of c mod n using private keys p and q. Most of the time there are four square roots of c mod n. Very rarely there are two square roots of c mod n. Now Alice needs to determine which of the roots corresponds to the original message.

Rabin Algorithm is sometimes referred to as a version of RSA algorithm. The security of this cryptosystem is based on the difficulty of the factorization problem. However, Rabin has many advantages over RSA. The encryption is much faster than RSA's, while the decryption speed is comparable with RSA's. It is proven to be as strong as factoring. If there are two square roots of *c: x* and *y* such that $x \neq n - y$, then there are non-trivial factors of *n* by computing GCD(*x*+*y*,*n*).

Ironically, this fact is also a major disadvantage of Rabin cryptosystem. It is easy to factor *n* if the two square roots of *c: x* and *y* such that $x \neq n - y$ are known. Using one of Rabin signature schemes or by some other means, Bob can ask Alice to decrypt ciphertext *c* and obtain the second root *y* with a probability ½. This is known as a chosen ciphertext attack. Another difficulty of Rabin cryptosystem is that Alice needs to figure out which square root corresponds to the message.

Both shortcomings of the Rabin can be addressed by adding redundant bits to the end of every message. One can also use zeros or any preset string of bits. These bits allow Alice to identify the correct square root. In addition, returning only the root corresponding to the encrypted message protects against the chosen ciphertext attack. There is still a possibility that even with redundancy, the Rabin machine would return an incorrect root. However, if enough redundant bits are used, the probability of this happening is very small. It is widely suggested that 64 bits is sufficient number of redundant bits [54]. In this case, the probability of an error is 2^{-64}.

3.3 Square Roots Modulo *n=pq*

The decryption step requires Alice to take square root modulo $n=pq$. It is a three steps process:

1) take square root c mod p. There are two square roots : x_1 and x_2, where $x_2=p-x_1$
2) take square root c mod q. There are two square roots : y_1 and y_2 where $y_2=q-y_1$
3) get four square roots m_1,m_2,m_3,m_4 of c (mod n) using Chinese Remainder Theorem (CRT) on pairs (x_1,y_1), (x_1,y_2), (x_2,y_1) and (x_2,y_2)

$$m_1 = x_1 q(q^{-1} \bmod p) + y_1 p(p^{-1} \bmod q) \pmod{n}. \tag{3.2}$$

$$m_2 = x_1 q(q^{-1} \bmod p) + y_2 p(p^{-1} \bmod q) \pmod{n}. \tag{3.3}$$

$$m_3 = x_2 q(q^{-1} \bmod p) + y_1 p(p^{-1} \bmod q) \pmod{n}. \tag{3.4}$$

$$m_4 = x_2 q(q^{-1} \bmod p) + y_2 p(p^{-1} \bmod q) \pmod{n}. \tag{3.5}$$

q ($q^{-1} \bmod p$) (mod n) and p ($p^{-1} \bmod q$) (mod n) can be precomputed at the time of key generation.

Another way to compute it is to find a and b satisfying $ap + bq = 1$, using the extended GCD algorithm. Then:

$$m_1 = apx_1 + bqy_1 \pmod{n}. \tag{3.6}$$

$$m_2 = apx_1 - bqy_1 \pmod{n}. \tag{3.7}$$

$$m_3 = -m_1 = n - m_1 \pmod{n}. \tag{3.8}$$

$$m_4 = -m_2 = n - m_2 \pmod{n}. \tag{3.9}$$

If primes p and q are Blum primes ($p \bmod 4 = 3$ and $q \bmod 4 = 3$), then the square roots from steps 1) and 2) are easy to compute:

$$x_1 = c^{\frac{p+1}{4}} \bmod p,\ x_2 = p - x_1, \tag{3.10}$$

$$y_1 = c^{\frac{p+1}{4}} \bmod p,\ y_2 = p - y_1 . \tag{3.11}$$

For non-Blum primes ($p \bmod 4 = 1$ and $q \bmod 4 = 1$) it is harder to compute the square roots. Even though it is possible to use non-Blum primes, it is much more practical to use Blum primes for the Rabin cryptosystem.

3.4 Extended Square Root Algorithm mod *p*

To extend the Rabin Cryptosystem to the domain of Gaussian integers, the square root algorithm was developed. As was already mentioned in Section 2.1, only the subset of Gaussian primes is considered: real primes p such that $p \bmod 4 = 3$ (Blum primes). The algorithm is below:

Algorithm 3.4.1 Extended Square root algorithm mod p

Given: $H=c+di=(c,d)$ – Gaussian integer

p – real Blum prime

Computing: $S=(a,b)$ square root of H mod p

(1) **if** ($d = 0$)

(2) $x := c^{(p+1)/4} \pmod p$;

(3) **if** ($x^2=c$) (mod p) // square root of c exists

(4) **return** { (x,0), (-x,0) };

(5) **else** // square root of c doesn't exist

(6) **return** { (0,x), (0,-x) };

(7) **endif**

(8) **else**

(9) $n := |(c,d)|^{(p+1)/4} \pmod{p}$;

(10) **if** $(n^2 \neq |(c,d)|) \pmod{p}$ // square root of $|H|$ doesn't exist

(11) **return** {}; // no square roots of H exists

(12) **else**

(13) $t := 2^{-1}(c+n) \pmod{p}$;

(14) $x := t^{(p+1)/4} \pmod{p}$;

(15) **if** $(x^2 = t) \pmod{p}$ // square root of t exists

(16) $a := x$;

(17) $b := (2a)^{-1} d \pmod{p}$;

(18) **return** $\{(a,b), (-a,-b)\}$;

(19) **else** // square root of t doesn't exist

(20) $b := x$;

(21) $a := (2b)^{-1} d \pmod{p}$;

(22) **return** $\{(a,b), (-a,-b)\}$;

(23) **endif**

(24) **endif**

(25) **endif**

Note that $2^{-1} \bmod p$ is simply $\frac{p+1}{2}$, since

$$2\frac{p+1}{2} = p+1 = 1 \pmod{p}. \tag{3.12}$$

A few simple theorems are needed to prove the validity of the algorithm and show how it was derived.

Theorem 3.4.1

H has a square root if and only if |*H*| has a square root (mod *p*)

Proof:

Suppose *G:* ord(*G*)=p^2-1 is a generator and $H=G^k \bmod p$.

$\Rightarrow$ It is known that *H* has a square root and the aim is to prove that |*H*| has a square root. If *H* has a square root *S* and $H=G^k \bmod p$ then *k* is divisible by 2. Looking at $|H| \bmod p$:

$$|H| = |G^k| = |G|^k \bmod p. \tag{3.13}$$

Since *k* is divisible by 2, the square root of |*H*| exists and equals $|G|^{k/2}$.

$\Leftarrow$ It is known that |*H*| has a square root and the aim is to prove that *H* has a square root. Since the square root of |*H*| exists, then *k* is divisible by 2. From this directly follows that square root of *H* exists.

Q.E.D.

Theorem 3.4.2

If $H=(c,0)$ then:

1) H always has a square root S.

2) If $a^2=c \bmod p$, then $(a,0)$ and $(-a,0)$ are the square roots of H.

3) If c does not have a square root, then $(0,b)$, $(0,-b)$ are the square roots of H, and

$$b^2 = -c \bmod p\,. \tag{3.14}$$

Proof:

1) It is true that

$$|H| = |(c,0)| = c^2 \pmod p\,. \tag{3.15}$$

This implies that the square root of $|H|$ exists and equals to c. According to Theorem 3.4.1**,** square root of H must exist also.

2) It is given than that $H=(c,0)$ and c has a square root. From 1) it follows that there is a square root $S=(a,b)$ of $H \bmod p$.

$$H=(c,0)=(a^2-b^2,\ 2ab) \text{ and } 2ab=0 \pmod p \tag{3.16}$$

From the identity

$$2ab = 0 \bmod p \tag{3.17}$$

follows that a or b must be 0. In addition, it is known that $c= a^2-b^2$ and c has a square root.

Suppose a=0 mod p, then c=-b^2. From this follows that c does not have a square root. This is a contradiction because it is known that c has a square root. Consequently, the only possibility is that b=0 mod p and c=a^2 mod p, thus (a,0) and (-a,0) are the only two square roots of H.

3) It is given that H=(c,0) and c does not have a square root. From 1) it follows that there is a square root S=(a,b) of H mod p.

$$H=(c,0)=(a^2-b^2,\ 2ab) \text{ and } 2ab=0 \pmod{p}. \tag{3.18}$$

From $2ab$ = 0 mod p follows that a or b must be 0. In addition, it is known that $c= a^2-b^2$ and c does not have a square root. Suppose b=0 mod p, then

$$c=a^2 \bmod p. \tag{3.19}$$

Therefore, c has a square root a mod p. This is a contradiction because it is known that c does not have a square root. Consequently, the only possibility is that a=0 mod p and c=-b^2 mod p. Thus (0,b) and (0,-b) are the only two square roots of H.

Q.E.D

Theorem 3.4.3

If $H=(c,d)$, $d \neq 0 \bmod p$ and r and $-r$ are square roots of $|H| \bmod p$, then:

1) There are exactly two square roots of $H \bmod p$: $S_1=(a,b)$, $S_2=(-a,-b)$.

   Neither a, nor b is 0.

2) Either

$$\{a = \sqrt{2^{-1}(c+r)} \text{ and } b = (2a)^{-1} d \pmod p\}. \tag{3.20}$$

or

$$\{b = \sqrt{-2^{-1}(c+r)} \text{ and } a = (2b)^{-1} d \pmod p\}. \tag{3.21}$$

Proof:

Since square roots of $|H|$ exist, the square root $S_1=(a,b)$ of $H \bmod p$ exists (Theorem 3.4.1**).**

$$S_1^2 = (a^2 - b^2, 2ab) = H \bmod p. \tag{3.22}$$

$S_2=(-a,-b)$ is also a square root of H because

$$S_2^2 = (-a,-b)^2 = (a^2 - b^2, 2ab) = H \pmod p. \tag{3.23}$$

$$\{b = \sqrt{-2^{-1}(c+r)} \text{ and } a = (2b)^{-1} d \pmod{p}\} . \tag{3.24}$$

$$d = 2ab \text{ and } d \neq 0 \pmod{p} \tag{3.25}$$

implies that neither *a,* nor *b* is 0. Moreover, $|S_1| = |S_2|$ must also equal to $r = a^2 + b^2$ or $r = -a^2 - b^2 \pmod{p}$. It is known that

$$a^2 - b^2 = c \bmod p . \tag{3.26}$$

If $r = a^2 + b^2$, then

$$\sqrt{2^{-1}(c+r)} = \sqrt{2^{-1}(a^2 - b^2 + a^2 + b^2)} = \sqrt{a^2} \pmod{p} . \tag{3.27}$$

$\sqrt{a^2} \pmod{p}$ is *a* or *–a.* For either *a* or *–a* find *b* using $d = 2ab$:

$$b = (2a)^{-1} d \pmod{p} . \tag{3.28}$$

If $r = -a^2 - b^2$, then

$$\sqrt{-2^{-1}(c+r)} = \sqrt{-2^{-1}(a^2 - b^2 - a^2 - b^2)} = \sqrt{b^2} \pmod{p} . \tag{3.29}$$

$\sqrt{b^2} \pmod p$ is b or $-b$. For either b or $-b$ find a using $d = 2ab$:

$$a = (2b)^{-1} d \pmod p . \tag{3.30}$$

Each of the possibilities $r = a^2 + b^2$ or $r = -a^2 - b^2 \pmod p$ yields two square roots. Also note that only one of $\sqrt{2^{-1}(c+r)}$ or $\sqrt{-2^{-1}(c+r)}$ exists because $2^{-1}(c+r)$ and $-2^{-1}(c+r)$ are opposites of each other. Consequently, there are exactly two square roots of H.

Q.E.D

Note that on lines (17) and (21) of Algorithm 3.4.1 the modular inverse $(2a)^{-1}d$ or $(2b)^{-1}d \pmod p$ is needed. Before this the condition needs to be checked to make sure that neither a nor b equals to 0 mod p. Fortunately, this is easy to do. If S is in the form of $(a,0)$ or $(0,b)$, then $H=S^2$ must be in the form of $(c,0)$. This means that the only condition needed to be checked is: if $d=0 \bmod p$. If $d=0 \bmod p$, then Theorem 3.4.2 is applied to compute the Gaussian square roots of $H=(c,0)$. Only one square root of a real integer is needed to do it. Lines (1)-(8) of Algorithm 3.4.1 handle to the case when $d=0 \bmod p$.

The rest of the algorithm (lines (8) to (25)) corresponds to the more general case when $d \neq 0 \bmod p$. The square root of $|H| \bmod p$ is taken on lines (13) and (14). According to Theorem 3.4.1, the execution can stop if there is no square root of $|H| \bmod p$ (lines (10) and (12)). Otherwise, it must be true that the Gaussian square roots of H exist and Theorem 3.4.3 is used to compute them.

The most computationally expensive operations in Algorithm 3.4.1 are the real integer square root operations. If H is in the form of $(a,0)$, only one real integer square root operation is required. Otherwise, two real integer square root operations and one modular inverse operation are required.

3.5 Extended Square Root Algorithm mod $n=pq$

The algorithm for finding square roots modulo $n= pq$ can be constructed by using Algorithm 3.4.1 together with CRT (Chinese Remainder Theorem).

Algorithm 3.5.1 Extended Square root algorithm mod $n=pq$

Given: p, q - real Blum primes

$n=pq$ - product of p and q

$H=(c,d) \mod n$ - Gaussian integer

Find: All $S_i=(a_i,b_i)$ *:* $S_i^2=H \pmod n$

Step 1. Find the square root $S_p=(a_p,b_p)$ *of* $H \pmod p$ using Algorithm 3.4.1. If S_p does not exist stop, there is no square root (mod n) for H.

Step 2. Find the square root $S_q=(a_q,b_q)$ *of* $H \pmod q$ using Algorithm 3.4.1. If S_q does not exist then stop, there is no square root (mod n) for H.

Step 3. Reconstruct all different S_i using CRT in the following way:

$S_1=(a_1,b_1)$, where

$$a_1=a_p \ q\ (q^{-1} \bmod p) + a_q \ p\ (p^{-1} \bmod q) \pmod n \qquad (3.31)$$

$$b_1=b_p\ q\ (q^{-1} \bmod p) + b_q\ p\ (p^{-1} \bmod q) \pmod n \tag{3.32}$$

$S_2=(a_2,b_2)$, where

$$a_2=-a_p\ q\ (q^{-1} \bmod p) + a_q\ p\ (p^{-1} \bmod q) \pmod n \tag{3.33}$$

$$b_2=-b_p\ q\ (q^{-1} \bmod p) + b_q\ p\ (p^{-1} \bmod q) \pmod n \tag{3.34}$$

$S_3=(a_3,b_3)$, where

$$a_3=-a_p\ q\ (q^{-1} \bmod p) - a_q\ p\ (p^{-1} \bmod q) \pmod n \tag{3.35}$$

$$b_3=-b_p\ q\ (q^{-1} \bmod p) - b_q\ p\ (p^{-1} \bmod q) \pmod n \tag{3.36}$$

$S_4=(a_4,b_4)$, where

$$a_4=a_p\ q\ (q^{-1} \bmod p) - a_q\ p\ (p^{-1} \bmod q) \pmod n \tag{3.37}$$

$$b_4=b_p\ q\ (q^{-1} \bmod p) - b_q\ p\ (p^{-1} \bmod q) \pmod n. \tag{3.38}$$

$$q\ (q^{-1} \bmod p) \pmod n \tag{3.39}$$

and

$$p\ (p^{-1} \bmod q) \pmod n \tag{3.40}$$

can be precomputed.

There could be at most four distinct S_i but it is possible to have one or two distinct S_i. Another way to compute it is to find a and b satisfying

$$ap + bq = 1, \tag{3.41}$$

using the extended GCD algorithm:

$$S_1 = (ap\, S_p + bq\, S_q) \pmod{n} \tag{3.42}$$

$$S_2 = (ap\, S_p - bq\, S_q) \pmod{n} \tag{3.43}$$

$$S_3 = -S_1 \pmod{n} \tag{3.44}$$

$$S_4 = -S_2 \pmod{n}. \tag{3.45}$$

3.6 Extended Rabin Cryptosystem

Using the extended square root algorithm (Algorithm 3.5.1) the following algorithm can be formulated:

Algorithm 3.6.1 Extended Rabin Cryptosystem

Key generation

Alice selects two distinct primes p and q and calculates $n=pq$. She publishes n as a public key.

Encryption

Given message $M = (m_1, m_2) : \ 0 \le m_1, m_2 \le n-1$, Bob computes the ciphertext

$$C = (c_1, c_2) = M^2 \bmod n = \left(\left(m_1^2 - m_2^2 \right) \bmod n, 2m_1 m_2 \bmod n \right) \tag{3.46}$$

and sends *C* to Alice.

Decryption

Given the ciphertext *C* Alice computes the square roots of *C* mod *n* using private keys *p* and *q* and Algorithm 3.5.1. Most of the time there are four square roots of *C* mod *n*. Very rarely there are two square roots of *C* mod *n*. Now Alice needs to determine which of the roots corresponds to the original message.

To use the original Rabin algorithm Bob would have to break the large messages into blocks $m_1, m_2, .. m_L$ such that $0 \le m_i \le n - 1$. With Algorithm 3.6.1, Bob would need to do the same thing. The only difference is that Bob would send two blocks at the time.

3.7 Security of the Extended Rabin Cryptosystem

It is clear that the Extended Rabin Algorithm (Algorithm 3.6.1) is as secure as the original Rabin algorithm (Algorithm 3.2.1). If the adversary can compute find two square roots of $C : S_1 = (a_1, b_1)$ and $S_2 = (a_2, b_2)$ such that $S_1 \ne -S_2$ he/she can find non-trivial factors of *n* by computing

$$\gcd(a_1 + a_2, n), \tag{3.47}$$

$$\gcd(b_1 + b_2, n), \tag{3.48}$$

$$\gcd(|a_1 - a_2|, n) \tag{3.49}$$

or

$$\gcd(|b_1 - b_2|, n) \tag{3.50}$$

$|a_1 - a_2|$ and $|b_1 - b_2|$ here are absolute values.

The Extended Rabin Algorithm, as the original, is vulnerable to a chosen ciphertext attack. In addition, there is still a problem of selecting the correct square root. As with the Original Rabin Algorithm, both problems can be addressed by adding preset bits to the end of every message. With Algorithm 3.6.1 the message M consists of two blocks m_1 and m_2. The redundant bits could be added to m_1, m_2 or both m_1 and m_2. The advantage of the Extended Rabin Algorithm is that only half as many bits per block are needed as with the Original Rabin Algorithm to achieve the same probability of returning the correct square root. For example, to achieve the probability of an error of 2^{-64}, 32 redundant bits per block are required. With the original 64 bits are required.

When the Rabin algorithm is used with redundant bits, the proof of equivalency to factoring is no longer valid. This means that Rabin Cryptosystem with redundant bits

may be easier to break than factoring. If this is the case, then Gaussian integers offer enhanced security because the order of Gaussian integers mod p is p^2-1 as opposed to p-1 with real integers. The order of Gaussian integers mod $n=pq$ is

$$\text{lcm}(p^2-1,q^2-1), \tag{3.51}$$

as opposed to

$$\text{lcm}(p-1,q-1) \tag{3.52}$$

with real numbers. Moreover, the fact that there are less redundant bits is also likely to increase security.

3.8 Chapter Summary

In this chapter, an extension of the Rabin cryptosystem into the field of Gaussian integers was formulated. The extended cryptosystem employs a new square root algorithm for Gaussian integers, which is presented in Section 3.4 and proven. The Extended Rabin cryptosystem is at least as secure as the original Rabin Cryptosystem. When used with redundant bits, it offers the advantage of using less number of bits.

CHAPTER 4

ANALYSIS OF RSA ALGORITHM OVER GAUSSIAN INTEGERS

4.1 Description of RSA Algorithm over the Field of Gaussian Integers

RSA is a widely used algorithm based on the difficulty of factoring a product of two primes. The original RSA algorithm over the field of real integers is presented below:

Algorithm 4.1.1 Original RSA algorithm

Key Generation: Generate two large distinct real primes p and q. Compute $n=pq$. Compute $\varphi(n) = (p-1)(q-1)$. Select a random integer e such that $1 < e < \varphi(n)$ and $\gcd(e, \varphi(n)) = 1$. Compute $d \coloneqq e^{-1} \bmod \varphi(n)$. The pair n and e is the public key, and d is the private key.

Encryption: Given a message m (represented as a real integer) compute the ciphertext $c \coloneqq m^e \bmod n$.

Decryption: Compute the original message $m \coloneqq c^d \bmod n$.

In [30], RSA was extended into the field of Gaussian integers. It is presented below:

Algorithm 4.1.2 RSA algorithm with Gaussian integers

Key Generation: Generate two large Gaussian primes P and Q. Compute $N=PQ$. Compute $\varphi(N)=\varphi(P)\varphi(Q)$. Select a random integer e such that $1<e<\varphi(N)$ and $\gcd(e,\varphi(N))=1$. Compute $d=e^{-1} \bmod \varphi(N)$. Pair N and e is a public key, and d is the private key.

Encryption: Given a message M (represented as a Gaussian integer) compute cipher text $C=M^e \text{ MOD } N$.

Decryption: Compute the original message $M=C^d \text{ MOD } N$.

4.2 Cryptanalysis of RSA Algorithm over the Field of Gaussian Integers

Algorithm 4.1.2 is the same as in [30]. The notation was converted and a special system introduced in [30] to avoid negative integers was omitted.

Suppose $N=PQ$, where P and Q are Gaussian primes. N is a public key known to everybody. If one can factor N, the cryptosystem is broken and it is possible to read all the messages. There are three possibilities:

1) $P=(p,0)$ and $|Q| \bmod 4=1$ where p is a real Blum prime
2) $|P| \bmod 4=1$ and $|Q| \bmod 4=1$
3) $P=(p,0)$ and $Q=(q,0)$ where p and q are real Blum primes.

The first possibility is clearly not secure. If $P=(p,0)$ and $Q=(a,b)$ then $N=(ap,bp)$. To determine the factors of N one needs to find $\gcd(ap,bp)=p$, where $\gcd(a,b)=1$, because Q is a Gaussian prime.

Example 4.2.1 RSA with Gaussian primes of mixed type (small numbers)

P=(23,0), Q=(9,4),

N=PQ=(23,0)(9,4)=(207,92)

gcd(207,92) = 23

Example 4.2.2 RSA with Gaussian primes of mixed type (larger numbers)

P=(289518848489460091577546380,0), Q=(51325165669337,1615288995535)

N=PQ=(14859602863117217452522427580290411050861 1,

467656609964989843358764006194662111960 5)

gcd(14859602863117217452522427580290411050861 1,

467656609964989843358764006194662111960 5)=

=289518848489460091577546380

It takes a fraction of a second to compute the factor of N in large prime example. Consequently, this combination of Gaussian primes should never be used in Algorithm 4.1.2**.**

In case 2), both P=(a,b) and Q=(c,d) are non-Blum Gaussian primes. The one-to-one relationship between real primes and non-Blum Gaussian primes could be used. If in Algorithm 4.1.2 both P and Q are non-Blum Gaussian primes, then, after converting the Gaussian integers into real integers, the resulting algorithm is the original RSA algorithm

over real integers. For the examples below the numerical examples from [30] were considered to illustrate the point.

Example 4.2.3 RSA with non-Blum Gaussian primes

The original example from [30] has the following:

Key generation: Suppose P=(533,162) and Q=(17,10) (4.1)

N=PQ=(7441,8084). (4.2)

$\varphi(N) = (|P|-1)(|Q|-1) = (310333\text{-}1)(389\text{-}1)=120408816$ (4.3)

Select e = 56852657. (4.4)

$d = e^{-1} \bmod \varphi(N) = 56852657^{-1} \bmod 120408816 = 98072417$ (4.5)

The public key is N = (7441,8084); e = 56852657

Encryption:

Suppose a plaintext M = (0,999)

$$C = M^e \text{ MOD } N = (0,999)^{56852657} \text{ MOD } (7441,8084) = \\ = (\text{-}1530,2765) \quad (4.6)$$

Decryption:

$$M = C^d \text{ MOD } N = (-1530,2765)^{98072417} = (0,999) \quad (4.7)$$

To get the equivalent of real integer RSA protocol the numerical representation of I has to be computed. If $N = a + bi$, then $a + bi = 0 \pmod{|N|}$, where $i := -ab^{-1} \bmod |N|$. For this example, $N = (7441,8084)$ or 7441+8084i.

$$i = (\text{-}7441)8084^{-1} = 120712096 * 83312011 = \\ = 90868181 \pmod{120719537} \quad (4.8)$$

To get the equivalent real integer RSA protocol the numerical representation of I has to be computed. If $N = a + bi$, then $a + bi = 0 \pmod{|N|}$, where $i \coloneqq -ab^{-1} \bmod |N|$.

For this example N=(7441,8084) or 7441+8084i.

$$\begin{aligned} i &= (\text{-}7441)8084^{-1} = 120712096 * 83312011 = \\ &= 90868181 \pmod{120719537} \end{aligned} \tag{4.9}$$

The equivalent real integer RSA protocol:

Key generation: Set $p = |P| = |(533,162)| = 310333,$ (4.10)

$$q = |Q| = |(17,10)| = 389 \tag{4.11}$$

$$n = |N| = |P||Q| = pq = |(7441,8084)| = 120719537. \tag{4.12}$$

$$\begin{aligned} \varphi(n) = \varphi(|N|) = \varphi(N) = (|P| - 1)(|Q| - 1) = (p-1)(q-1) = \\ = (310333 - 1)(389 - 1) = 120408816 \end{aligned} \tag{4.13}$$

Using the same keys e and d:

$$e = 56852657 \tag{4.14}$$

$$d = e^{-1} \bmod \varphi(N) = 56852657^{-1} \bmod 120408816 = 98072417 \tag{4.15}$$

Here the public key is n = 120719537; e = 56852657

Encryption:

With the Gaussian integer protocol, the message is M = (0,999) = 999i.

Convert M to m as follows:

$$m = 999i = 999 * 90868181 = 116940532 \bmod 120719537 \tag{4.16}$$

In the Gaussian integer protocol:

$$\begin{aligned} C &= M^e \text{ MOD } N = \\ &= (0,999)^{56852657} \text{ MO}D\ (7441,8084) = (\text{-}1530, 2765) \end{aligned} \tag{4.17}$$

There are several ways to get the corresponding c:

1) Convert C to c using the conversion algorithm (Algorithm 2.1.1):

$$c = -1530 + 2765i = -1530 + 2765 * 90868181 = \\ = 33162438(\text{mod } 120719537) \tag{4.18}$$

2) Use the integer exponentiation algorithm:

$$c = m^e \bmod n = \\ = 116940532^{56852657} = 33162438 \text{ (mod } 120719537) \tag{4.19}$$

Note that 33162438 MOD (7441,8084) = (-1530,2765) (4.20)

Decryption:

With the Gaussian integer protocol the message is:

$$M = C^d \text{ MOD } N = (-1530, 2765)^{98072417} = (0, 999) \tag{4.21}$$

As with cipher text *c*, there are several ways to get the corresponding *m*:

1) Convert *M* to *m* using the conversion algorithm (Algorithm 2.1.1):

$$m = 999i = 999*90868181 = 116940532 \text{ (mod } 120719537) \tag{4.22}$$

2) Use the integer exponentiation algorithm:

$$m = c^d \bmod n = \\ = 33162438^{98072417} = 116940532 \text{ (mod } 120719537) \tag{4.23}$$

Note that 116940532 MOD (7441,8084) = (0,999) (4.24)

From Example 4.2.3 several facts are clear:

- If the adversary can break real integer RSA then he/she automatically can automatically break the corresponding Gaussian integer RSA. Consequently, when two non-Blum Gaussian primes are used for Algorithm 4.1.2 there is no added security.
- Note that the cipher text with Gaussian RSA and the corresponding real integer RSA has the same number of digits (-1530+2765*i* vs. 33162438). This is not

surprising because the one-to-one relationship implies that on average the number of digits in Gaussian integers and the corresponding real integers would be the same. The argument that Gaussian integer RSA packs more information is wrong. In fact, if any of the message representation schemes are used (like redundancy) the amount of information packed into each message would be less with Gaussian RSA.

- There is no need for "domain of validity" concept as described in [30]. The Gaussian integer modulo operation, if defined carefully, is not ambiguous. Negative values represent information. In fact, if the "domain of validity" is used, less information is packed into each message.

It was demonstrated that, if two non-Blum Gaussian primes are used in Algorithm 4.1.2, then there is one-to-one correspondence to real integer RSA. From this, one could derive a conclusion that the security of Gaussian integer RSA with two non-Blum primes is the same as with real integer RSA. However, this may not be the correct conclusion. In fact, it is likely that Gaussian integer RSA is less secure than the corresponding real integer RSA.

The reason for this is that the problem of representing $n=pq$ (p and q are large primes) as a sum of two integer squares is a hard problem when factors p and q are large and unknown. If n can be represented as $n = a^2 + b^2$ and $n = c^2 + d^2$ where $c \neq n - a$ and $c \neq n - b$, then n can be factored easily by doing the following:

1) Multiply $a+bi$ by $c+di$ modulo n

$$(a+bi)(c+di) = e+fi \pmod{n} \tag{4.25}$$

2) Compute gcd(*e*,*f*). gcd(*e*,*f*) would equal to either *p* or *q*.

If representing *n* as a sum of two squares were an easy problem, then the factoring of *n* would be an easy problem also. By using Algorithm 4.1.2 with two non-Blum Gaussian primes, a partial solution to the factoring problem is given away and, possibly, the entire solution.

The method described above is a generalization of a method for factorization used by Fermat. It is based on an idea that, if there are known two integers *x* and *y* so that $x^2 \equiv y^2 (\mathrm{mod}\, n)$ holds, then it can be used for factoring of *n*=*pq*. Details of the algorithm and proof are provided in [32] .

Now the case when Blum Gaussian primes are used in Algorithm 4.1.2 is discussed. When primes *P* = (*p*,0) and *Q* = (*q*,0) (*p* and *q* are Blum real primes) are used in Algorithm 4.1.2, it becomes:

Algorithm 4.2.1 RSA algorithm with Gaussian integers and Blum Gaussian primes

Key Generation: Generate two large real primes *p* and *q*. Compute *n*=*pq*. Compute $\varphi(n) = (p^2-1)(q^2-1)$. Select a random integer *e* such that $1 < e < \varphi(n)$ and $\gcd(e, \varphi(n)) = 1$. Compute $d = e^{-1} \bmod \varphi(n)$. Pair *n* and *e* is a public key, and *d* is the private key.

Encryption: Given a message $M = (m_1, m_2)$, where $1 \le m_1 < n$ and $1 \le m_2 < n$, compute cipher text $C := M^e \bmod n$. Here "mod" operation on a Gaussian integer is as follows: if *G*=(*a*,*b*), then

$$G \bmod n = (a,b) \bmod n = (a \bmod n, b \bmod n) \tag{4.26}$$

Decryption: Compute the original message $M = C^d \bmod n$.

This algorithm is described in [28] and it is very similar to real integer RSA. However, there are differences. In Algorithm 4.2.1, e and d range from 1 to $\varphi(n) = (p^2 - 1)(q^2 - 1)$, as opposed to $\varphi(n) = (p-1)(q-1)$ in Algorithm 4.1.1. The order of a Gaussian integer modulo $n = pq$ (p and q are Blum real primes) is much larger than the order of a real integer modulo n. In fact,

$$\text{ord}(G) \bmod n \le \text{lcm}(p^2 - 1, q^2 - 1) \tag{4.27}$$

as opposed to

$$\text{ord}(g) \bmod n \le \text{lcm}(p-1, q-1) \tag{4.28}$$

where G is a Gaussian integer, g is a real integer.

The primes p and q could be selected such that ord(G) mod n would equal to

$$\frac{(p^2 - 1)(q^2 - 1)}{24} \tag{4.29}$$

However, larger order does not necessarily mean greater security for the RSA.

Currently, RSA is not proven to be as secure as factoring, although many scientists believe that it is likely the case. If it is the case, then Algorithm 4.2.1 is as secure as Algorithm 4.1.2. On the other hand, if breaking the RSA is not as hard as factoring, then it is possible that Gaussian integers add security to the RSA.

At first glance, it seems that the message in Algorithm 4.2.1 packs more information than the message in Algorithm 4.1.2**.** However, it is incorrect. Sending one message with Algorithm 4.2.1 is equivalent to sending two messages with Algorithm 4.1.1 as far as network bandwidth is concerned. Moreover, it takes longer to encrypt a Gaussian integer message than to encrypt two real integer messages. In Algorithm 4.2.1, in order to encrypt or decrypt, the following operation has to be performed:

$$G^k \bmod n, \tag{4.30}$$

where G=(a,b) is a Gaussian integer, and $1 < k < (p^2 - 1)(q^2 - 1)$.

It takes three real integer multiplications and several real integer additions to do one the Gaussian integer multiplication. It takes two real integer multiplications and several real integer additions to do one the Gaussian integer square. The integer multiplication is much more time consuming than the integer addition so the additions will be ignored in the subsequent analysis. When using the square-and-multiply algorithm to perform Gaussian exponentiation, the average running time is

$$t_1 = 3.5 t_m \log_2 \frac{(p^2 - 1)(q^2 - 1)}{2} \tag{4.31}$$

where t_m is the time required for multiplication of two real integers.

To encrypt or decrypt a message with the real integer RSA the following operation has to be performed:

$$g^m \bmod n\,, \tag{4.32}$$

where g is a real integer and $1 < m < (p-1)(q-1)$

The time required to perform two integer exponentiation operations is:

$$t_2 = 3t_m \log_2 \frac{(p-1)(q-1)}{2} \tag{4.33}$$

No additions are necessary for real integer exponentiation operation. Clearly, $t_1 > t_2$. It takes less time to encrypt or decrypt two real integer messages than one Gaussian integer message. Consequently, Algorithm 4.2.1 does not have any advantages as far as encryption or decryption time of a given amount of data is concerned. The example below demonstrates this point.

Example 4.2.4 Algorithm 4.2.1 vs. Algorithm 4.1.1

Key generation: Suppose p=251, q=263

$$n = pq = 66013 \tag{4.34}$$

$$\varphi(n) = (p^2-1)(q^2-1) = (251^2\text{-}1)(263^2\text{-}1)=4357584000 \tag{4.35}$$

Choose e = 56852657, then

$$d = e^{-1} \bmod \varphi(N) = 56852657^{-1} \bmod 4357584000 = 1716163793 \tag{4.36}$$

The public key is n = 66013;e = 56852657

Encryption:

Let message $M = (m_1, m_2)$ = (55555,44444)

$$C = (c_1, c_2) = M^e \bmod n = (55555, 44444)^{56852657} \bmod 66013 = \\ = (31754,12046) \tag{4.37}$$

Decryption:

$$M = C^d \text{ MOD } N = \\ = (31754,12046)^{1716163793} \bmod 66013 = (55555, 44444) \tag{4.38}$$

The corresponding real integer RSA:

Key generation: p=251, q=263

$$n = pq = 66013$$

$$\varphi(n) = (p-1)(q-1) = (251-1)(263-1) = 65500 \tag{4.39}$$

To get e, reduce $e \bmod \varphi(n)$ or mod 65500:

$$e = 56852657 \bmod 65500 = 64157 \tag{4.40}$$

To get d, reduce $d \bmod \varphi(n)$ or mod 65500:

$$d = 1716163793 \bmod 65500 = 63793 \tag{4.41}$$

Note that 63793*64157=1 mod 65500

The public key is n = 66013; e = 64157

Encryption:

In the corresponding Gaussian RSA, the message was $M=(m_1,m_2)$=(55555,44444)

Encrypt m_1 and m_2 separately as follows:

$$c_1 = m_1^e \bmod n = 55555^{64157} \bmod 66013 = 61927 \tag{4.42}$$

$$c_2 = m_2^e \bmod n = 44444^{64157} \bmod 66013 = 22993 \quad (4.43)$$

Decryption:

Decrypt m_1 and m_2 separately as follows:

$$m_1 = m_1^d \bmod n = 61927^{63793} \bmod 66013 = 55555 \quad (4.44)$$

$$m_2 = m_2^d \bmod n = 22993^{63793} \bmod 66013 = 44444 \quad (4.45)$$

Example 4.2.4 illustrates that Algorithm 4.2.1 and Algorithm 4.1.1 have approximately the same performance when encrypting and decrypting the same amount of data. In fact, as was proved before, the original RSA over real integers would be slightly faster.

The extension of RSA the algorithm into the field of Gaussian integers (Algorithm 4.2.1) is viable only if real primes $p : p \bmod 4 = 3$ are used (Algorithm 4.2.1). The extended algorithm could add security only if breaking the original RSA is not as hard as factoring. Even in this case, it is not clear whether the extended algorithm would increase security. The Gaussian integer RSA is slightly less efficient than the original; therefore, the original real integer RSA is more practical.

4.3 Chapter Summary

In this chapter, it is shown that the extension of the RSA algorithm into the field of Gaussian integers is viable only when real primes p: p mod 4 = 3 are used. The extended algorithm could add security only if breaking original RSA is not as hard as factoring.

Even in this case, it is not clear if the extended algorithm would increase security. The Gaussian integer RSA is slightly less efficient than the original; therefore, the original real integer RSA may be more practical.

CHAPTER 5

A PSEUDO-RANDOM PIXEL REARRANGEMENT ALGORITHM BASED ON GAUSSIAN INTEGERS FOR IMAGE WATERMARKING

5.1 Algorithm Introduction

Steganography is a process of hiding information in a medium in such a manner that no one except the anticipated recipient knows of its existence ([61]). The history of steganography can be traced back to around 440 B.C.E, where the Greek historian Herodotus described in his writings about two events: one used wax to cover secret messages, and the other used shaved heads. With the explosion of internet as a carrier for various digital media, many new directions of this state-of-the-art emerged.

A notable application of steganography is watermarking of digital images, which is a useful tool for identifying the source, creator, owner, distributor, or authorized consumer of a document or an image. It has become very easy nowadays to copy or distribute digital images (whether copyrighted or not). A watermark is a pattern of bits inserted into a digital media for copyright protection ([12]). There are two kinds of watermarks: visible and hidden. A good visible watermark must be difficult for an unauthorized person to remove and can resist falsification. Since it is relatively easy to embed a pattern or a logo into a host image, the authorized person must make sure the visible watermark was indeed the one inserted by the author. In contrast, a hidden watermark is embedded into a host image by some sophisticated algorithm and is invisible to the naked eye. It could, however, be extracted by a computer.

There are many innovating watermarking algorithms and many more get published every day (such as recently published [3, 41, 53, 70]). In many image watermarking algorithms, for example in [24, 69, 72, 73], it is required to rearrange the pixels as a part of watermarking process. Randomness is desired during this step.

Modular arithmetic and, specifically, the integer exponentiation modulo prime numbers are widely used in modern cryptographic algorithms. One important property of integer exponentiation modulo prime is that it generates a sequence of integers that looks very much like a sequence of random numbers. This is a property that is desirable for pixel rearrangement algorithms. In this dissertation, the rearrangement step of watermarking algorithms is revisited and a different universal method for doing it is described. It is easy to replace rearrangement step in [24, 69, 72, 73] with the method described in this chapter. Moreover, this method can be used with most picture watermarking algorithms to enhance them.

One can look at Gaussian integers as an extension of real integers into two dimensions. They exhibit similar properties as regular integers but have some notable differences, that could be exploited in various fields, such as cryptography [27, 28, 30, 65]. One important difference is that they have a larger order for the same prime size, which provides the increased security.

In [69, 72], Arnold's cat map ([10]) was used to rearrange pixels for improving the performance of watermarking techniques. Here a replacement is described, namely, a novel pixel rearrangement algorithm based on Gaussian integers, to rearrange pixels in an image. It is demonstrated that the new algorithm is superior to Arnold's cat map in both time complexity and security. This technique is not a watermarking algorithm by itself

but rather a universal enhancement to any existing watermarking algorithms. The technique tends to increase robustness to noise by uniformly distributing noise throughout the image. The increase in robustness depends on the watermarking algorithm enhanced by the technique.

5.2 Proposed Pixel Rearrangement Algorithm

In this section the algorithms for pixel rearrangement are introduced and their computational complexity analyzed.

Algorithm 5.2.1 Pixel rearrangement based on Gaussian integers

Given*:* Image $I = (x, y)$ of size $m \times n$*;*

Output: Image $I' = (x', y')$ of size m × n;

1. Generate a prime $p > \max(m, n)$ and $p \bmod 4 = 3$.
2. Find a Gaussian integer generator $G = (a, b) \bmod p$, using Algorithm 2.10.1 or Algorithm 2.10.2.
3. Generate a random number s, such that $0 < s < p^2 - 1$.
4. $S = (s_x, s_y) := G^s \bmod p$ (5.1)
5. **while** ($s_x > m$ or $s_y > n$)
6. $S := SG \bmod p$
7. **end-while**
8. $C = (c_1, c_2) := S$

9. **for** i=1 **to** m

10. **for** j=1 **to** n

11. $I'\{c_1, c_2\} := I\{i, j\}$ (5.2)

12. $C := CG \bmod p$ (5.3)

13. **while** $c_1 > m$ **or** $c_2 > m$

14. $C := CG \bmod p$ (5.4)

15. **end-while**

16. **end-for**

17. **end-for**

Note that the last value of $C = (c_1, c_2)$ needs to be saved in order to rearrange back the pixels. Without the value of *C*, pixels could be rearranged back; however, it would require additional computation.

Algorithm 5.2.2 Reverse of Algorithm 5.2.1

1. $C_r := C$ (5.5)

2. **for** i=m **downto** 1

3. **for** j=n **downto** 1

4. $I\{i, j\} := I'\{c_1, c_2\}$ (5.6)

5. $C_r := C_r G^{-1} \bmod p$ (5.7)

6. **while** ($c_1 > m$ **or** $c_2 > m$)

7. $$C_r := C_r G^{-1} \bmod p \tag{5.8}$$

8. **end-while**

9. **end-for**

10. **end-for**

The time complexity of Algorithm 5.2.1 and Algorithm 5.2.2 can be defined in terms of p. The most computationally expensive operations of the algorithm are (5.1), (5.7) and (5.8). Suppose that u is the time spent to multiply two integers of size p. Assuming the square-and-multiply algorithm is used for exponentiation and Algorithm 1.3.1 is used to multiply two Gaussian integers, the time complexity of (5.1) is approximately:

$$3.5u \log_2 (p^2 - 1) \approx 7u \log_2 p \ . \tag{5.9}$$

Because the order of Gaussian integers is $p^2 - 1$, in *Step* 4 of Algorithm 5.2.1, $p^2 - 1$ multiplications are performed. Therefore, the number of multiplications required is:

$$\mathrm{O}\left(3u\left(p^2 - 1\right)\right) = \mathrm{O}\left(3up^2\right). \tag{5.10}$$

The total time complexity of Algorithm 5.2.1 is:

$$\mathrm{O}\left(3u\left(p^2 - 1\right) + 7u \log_2 p\right) = \mathrm{O}\left(up^2\right) \tag{5.11}$$

The complexity of integer multiplication u depends on the size of p. For small p, the most efficient algorithm is the naïve multiplication with time complexity of $O(l^2)$, where $l = \log_2 p$ is the size of p in bits. For a larger p, the multiplication algorithm in [43] is faster than the naïve method. The time complexity of the Karatsuba multiplication is $O(3l^{1.585})$. For an even larger p, the Toom-Cook (or Toom-3) algorithm is more efficient with a time complexity of $O(n^{1.465})$ [44]. The thresholds for the size of p vary widely with implementation details. However, it is reasonable to assume that most images would not be sufficiently large for the Toom-Cook or Karatsuba multiplication. Therefore, it can be assumed that the naïve multiplication method can be used and (5.11) becomes:

$$O\left(up^2\right) = O\left[\left(p \log_2 p\right)^2\right]. \tag{5.12}$$

This is the time complexity of Algorithm 5.2.1. The time complexity for Algorithm 5.2.2 is the same.

To minimize the time complexity, it is reasonable to select p close to $\max(m,n)$. If p is selected in such a way, then the time complexity in terms of image size is

$$O\left\{\left[\max\left(m,n\right)\log_2\left(\max\left(m,n\right)\right)\right]^2\right\}. \tag{5.13}$$

The rearrangement algorithm described above is universal and can be used for many purposes. It can be applied for image watermarking as follows:

Algorithm 5.2.3 Watermarking with pixel rearrangement based on Gaussian integers

1. Rearrange the image using Algorithm 5.2.1;

2. Apply the desired watermarking technique to the resulting rearranged image from Step 1;

3. Apply Algorithm 5.2.2 to the resulting image from Step 2.

Algorithm 5.2.4 Extraction of the watermark applied with Algorithm 5.2.3

1. Rearrange the image using Algorithm 5.2.1.

2. Extract the watermark using the watermarking extraction technique in Algorithm 5.2.2.

Note that in Algorithm 5.2.2, depending on watermarking technique, it may be possible to extract watermark and perform rearrangement on the watermark rather than on the image.

5.3 Cryptoimmunity of the Rearrangement Algorithm

From the properties of Gaussian integer group, it can be estimated how hard it is for an adversary to obtain the original image from the rearranged one. The less an adversary knows about the algorithm and parameters, the harder it is to determine the original arrangement. It is reasonable to look at the following three cases:

Case 1. The adversary knows nothing about the rearrangement algorithm used, but he/she suspects that some kind of an algorithm has been used. In this case, it is extremely hard for an adversary to figure out the original arrangement because there are too many possibilities. That is, there are $n!$ possible permutations; where n is the number of pixels in the image.

Case 2. The adversary knows that Algorithm 5.2.1 was used, but he/she does not know the parameters such as prime p, generator G, or private key s. In this case, the number of possible permutations for an image I of size $m \times n$ is:

$$\left(p^2-1\right)\left[\varphi\left(p^2-1\right)\right], \tag{5.14}$$

where φ is the Euler's totient function ([2]).

The formula (5.14) does not include the complexity of guessing p. The reason for this is that it is too computationally expensive to use a large p (refer to (5.12)). For efficiency, p should be close to the image size. The prime p in (5.14) can be selected in such a way that $\varphi\left(p^2-1\right)$ is maximized. To do this, one can select a prime with large prime divisors of $p+1$ and $p-1$. For example,

$$p+1=s_1q_1 \tag{5.15}$$

and

$$p-1=s_2q_2, \tag{5.16}$$

where s_1 and s_2 are small integers, and q_1 and q_2 are primes close to p in size. In this case:

$$\varphi\left(p^2-1\right)=\varphi\left(\left(p-1\right)\left(p+1\right)\right)=\varphi(s_1s_2)(q_1-1)(q_2-1) \tag{5.17}$$

and

$$o\left(\varphi\left(p^2-1\right)\right)=o\left((q_1-1)(q_2-1)\right)=o(q_1q_2)=o(p^2) \tag{5.18}$$

Consequently, the approximate computational complexity of (5.14) is:

$$o\left(\left(p^2-1\right)\left[\varphi\left(p^2-1\right)\right]\right)=o(p^4)=o\left(\max\left(m,n\right)^4\right) \tag{5.19}$$

Case 3. The adversary knows Algorithm 5.2.1 used, prime *p*, and a generator *G*. In this case, the number of possible permutations is limited to

$$p^2-1. \tag{5.20}$$

While it may be unreasonable to assume that the adversary would not know Algorithm 5.2.1, there is no reason to make a prime *p* and a generator *G* known. Therefore, case 2 may be the most reasonable security estimate.

If increased protection is desired, Algorithm 5.2.1 could be applied several times on the same image. Suppose that Algorithm 5.2.1 was applied t times on image I of size $m \times n$. In this case, the number of possible permutations is:

$$o\left(\max\left(m,n\right)^{4t}\right), \tag{5.21}$$

while the time to compute the rearranged image would still be reasonable and be on the same order in terms of image size:

$$\mathrm{O}\left\{t\left[\max\left(m,n\right)\log_2 \max\left(m,n\right)\right]^2\right\} = \mathrm{O}\left\{\left[\max\left(m,n\right)\log_2 \max\left(m,n\right)\right]^2\right\}. \tag{5.22}$$

Therefore, one can achieve the desired level of security by increasing the time it takes to rearrange the image somewhat. Multiple rearrangements could provide a desirable and practical tradeoff.

5.4 Comparison to Arnold's Cat Map Chaos Transformation

The Arnold's cat map transformation variation used in [69] is defined as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ l & l+1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \bmod N, \tag{5.23}$$

where N is the width of the square image. The possible values of l in (5.23) are $l : 1 < l < N - 2$. Therefore, the number of the transformations required is O(N). It is reasonable to assume that N is small enough to call for the naïve multiplication algorithms. Thus, the multiplication time complexity is

$$\mathrm{O}(\log_2^2 N), \tag{5.24}$$

and it has to be performed for every pixel (i.e., N^2 times). Therefore, the time complexity of Arnold's Cat Map is:

$$\mathrm{O}(N^3 \log_2^2 N). \tag{5.25}$$

Formula (5.25) should be compared with (5.13), assuming $N \approx \max(m,n)$. It is obvious that the computational complexity of Algorithm 5.2.1 described by (5.13) is much better than that of Arnold's Cat map described by (5.25).

As far as security, it is obvious that there are only $\mathrm{o}(N)$ possible permutations because $l : 1 < l < N - 2$. It is much smaller than $\mathrm{o}\left(\max\left(m,n\right)^4\right)$ for Algorithm 5.2.1.

Another important advantage of Algorithm 5.2.1 is that the transformed image does not have any visible patterns. After rearrangement with this the algorithm, the resulting image looks like random noise. The transformation with Arnold's Cat map, on the other hand, preserves visible patterns. Figure 5.1 clearly illustrates this point. At every step of Arnold's Cat map transformation, C1-C7 patterns are clearly visible. The

image B, on the other hand, looks like random noise. Consequently, Algorithm 5.2.1, when used for watermarking, is far superior to Arnold's Cat map in terms of security and computational time.

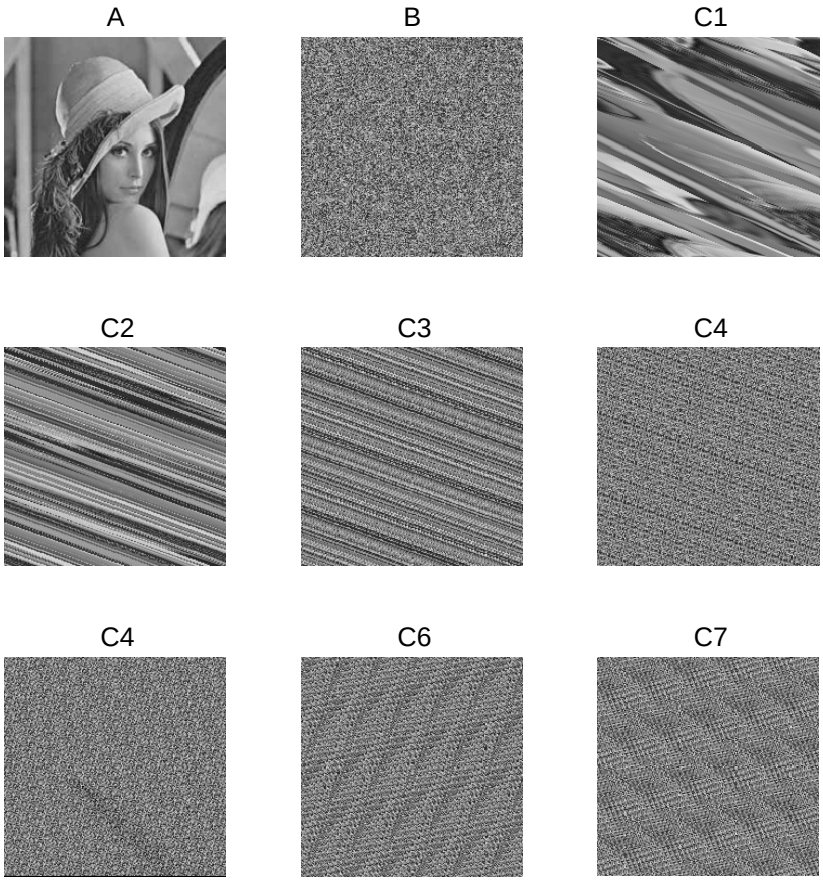

Figure 5.1 Image rearranged by Algorithm 5.2.1 and Arnold's Cat map side-by-side. A is the original image, B is the rearranged image by Algorithm 5.2.1, and C1-C7 are the steps of Arnold's Cat map rearrangement.

5.5 Example in Image Watermarking

Algorithm 5.2.1 can be used with general watermarking techniques. The following example illustrates its use of applying LSB substitution for watermark. Even though this technique does not provide a robust watermark, the use of rearrangement does improve the security by making the watermark virtually undetectable. When pixel rearrangement is used and the adversary looks at the last two bits of the watermarked image, all he/she sees is random noise. The only way to see the watermark is to rearrange the pixels.

Figure 5.2 illustrates the advantages of using the rearrangement algorithm for image watermarking. In Figure 5.2, (a) is the original Cameraman image, (b) is the two most significant bits of the Lena image to used as the watermark, (c) is the rearranged image of Cameraman using Algorithm 5.2.1, (d) is the watermarked image of the rearranged image using LSB substitution, (e) is the rearranged back of the preceding watermarked image using Algorithm 5.2.2, (f) is the extracted two bits of LSB, and (g) is the rearranged back of the preceding extracted image using Algorithm 5.2.2. Note that image (g) is exactly the same as the original watermark (b).

If the watermarking is performed without rearrangement, then the hidden watermark is easily detectible. By using the proposed algorithms, it is impossible to see the original watermark in image (f), which is random noise just like images (c) and (d). It is fairly difficult for the adversary to extract the original watermark, even though her/she knows that the watermark is hidden there. With sequential applications of Algorithm 5.2.1, the security could be enhanced to an arbitrary level, making watermark practically impossible to reconstruct for the adversary.

Figure 5.2 (a) The original Cameraman image, (b) the two most significant bits of Lena as the watermark, (c) the rearranged image of Cameraman using Algorithm 5.2.1, (d) the watermarked image of the rearranged image using LSB substitution, (e) the rearranged back of the preceding watermarked image using Algorithm 5.2.2, (f) the extracted two bits of LSB (g) the rearranged back of the preceding extracted image using Algorithm 5.2.2.

5.6 Chapter Summary

In this chapter, a new method of rearranging image pixels for watermarking based on the properties of Gaussian integers is described. It results in a random-looking image transformation that significantly improves the security of the embedded watermark. Moreover, it is much faster when compared to Arnold cat map. The proposed algorithm is an easy-to-implement practical technique that would enhance the security of any watermarking algorithm. It is flexible enough to offer variable levels of security.

CHAPTER 6

CONCLUSION

The application of Gaussian integers for DLP based public key cryptosystems was discussed. It was demonstrated that cryptosystems that are based on non-Blum Gaussian primes (primes $P=(a,b):|P|$ is a prime) are equivalent to real integer cryptosystems modulo $|P|$ (Algorithm 2.1.1 and Algorithm 2.1.2). Therefore, such cryptosystems do not offer any advantages over real integer cryptosystems. On the other hand, the cryptosystems based on Blum Gaussian primes (primes $P=(p,0):p$ is a prime) offer a longer cycle.

It was shown that the Gaussian integer DLP is substantially harder then the real integer DLP. Moreover, when solving the Gaussian integer DLP, one is required to solve two problems:

1) Lucas Sequences DLP with $Q \equiv 1 \bmod p$ (Theorem 2.5.2).

2) Real integer DLP.

The fact that these two problems seem to be very different, bodes very well for cryptography algorithms based on the Gaussian integer DLP. The solution of one problem may not lead to the solution of the other, so Gaussian integers offer additional protection.

In addition to allowing for assessing the complexity of the Gaussian integer DLP, Theorem 2.5.2 is the basis for Algorithm 2.8.1 (Lucas sequence Exponentiation of Gaussian integers (LSEG)). The LSEG algorithm achieves about 35% theoretical improvement in CPU time over real integer exponentiation. Under an implementation

with GMP 5.0.1 library it outperformed the GMP's "mpz_powm" function (the particularly efficient modular exponentiation function that comes with GMP library) by 40% for bit sizes 1000-4000, because of low overhead associated with LSEG. Moreover, some steps of the LSEG algorithm could be run in parallel (such version of the LSEG algorithm was denoted as LSEG*). LSEG* offers about 50% improvement over real integer exponentiation.

In this dissertation, the properties of Gaussian integers under modular multiplication and exponentiation were explored. Specifically, the order of Gaussian integers and its relationship to their norm was analyzed. Based on the relationship between the order and the norm, an efficient and practical algorithm to find generators for the Gaussian integer DLP cryptosystems was designed, namely, Algorithm 2.10.2.

In addition to DLP based cryptosystems, the factoring based cryptosystems with Gaussian integers were considered (i.e., RSA and Rabin). The Extended Square Root algorithm for Gaussian integers was derived and its validity proved. Using this algorithm the Rabin Cryptography algorithm was extended into the field of Gaussian integers. The resulting Extended Rabin Cryptography algorithm requires only half as many redundant bits as the original.

The analysis was performed on the extension of RSA into the domain of Gaussian integers. It yielded several interesting results, namely, that Gaussian primes $P = (a,b), b \neq 0$ do not offer any immediately tangible advantages over real primes and that the viability of Gaussian integer RSA is questionable.

Finally, a novel algorithm to rearrange the image pixels for image watermarking was derived. The new algorithm is much more efficient than Arnold's Cat map and it

provides a degree of cryptoimmunity to the watermarks. The proposed method can be used with most picture watermarking algorithms to enhance them.

The work presented in this dissertation can be extended in many directions including:

1. Improving the running time of LSEG (Algorithm 2.8.1)
2. Improving the performance of extended Rabin cryptosystem
3. Improving the security of the pixel rearrangement algorithm (Algorithm 5.2.1)

There are many other ways to extend research, but the abovementioned points seem to be the most promising.

Any improvement to the LSEG algorithm would mean an improvement in the running time of the Gaussian integer DLP based cryptosystems. Arguably, there is a lot of room for improvement. The slowest operation in the algorithm is the computation of Lucas sequences. Any improvement to the computation time of Lucas sequences would improve the performance of LSEG. The analysis in this dissertation used the algorithm published in [74]. It is analogous to square-multiply exponentiation for real integers. The algorithms published in [18] and [68] improve the running time of [74], however, it can probably be improved further. Moreover, any improvement to real integer exponentiation algorithms would improve the performance of LSEG.

The extended Rabin cryptosystem with Gaussian integers is not faster than real integer for the same amount of data. It is likely that the increase in number of dimensions in this case could be beneficial i.e., the extended Rabin algorithm with quaternions could

be faster than the original, provided that the square root for quaternions can be done with less than four integer exponentiations.

In all probability, the pixel rearrangement algorithm (Algorithm 5.2.1) can be modified to provide for greater cryptoimmunity with the same or almost the same efficiency.

REFERENCES

[1] A. M. AbdelFattah, A. El-Din, and H. M. A. Fahmy, "Efficient Implementation of Modular Multiplication on Fpgas Based on Sign Detection," in Design and Test Workshop (IDT), 2009, pp. 1-6.

[2] M. Abramowitz, and I. A. Stegun, *Handbook of Mathematical Functions*, New York: Dover, 1964.

[3] H. Al-Qaheri, A. Mustafi, and S. Banerjee, "Digital Watermarking Using Ant Colony Optimization in Fractional Fourier Domain," *Journal of Information Hiding and Multimedia Signal Processing,* vol. 1, no. 3, pp. 220-240, 2010.

[4] W. Alexi, B. Chor, O. Goldreich *et al.*, "Rsa and Rabin Functions: Certain Parts Are as Hard as the Whole," *Siam J. Comput.,* vol. 17, no. 2, pp. 194-209, 1988.

[5] Z. M. Ali, M. Othman, M. Said *et al.*, "Implementation of Parallel Algorithms for Luc Cryptosystem," in Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2008. SNPD '08. Ninth ACIS International Conference on, 2008, pp. 726-731.

[6] Z. M. Ali, M. Othman, M. Said *et al.*, "Computation of Private Key for Luc Cryptosystem," in Electrical Engineering and Informatics, 2009. ICEEI '09. International Conference on, 2009, pp. 418-422.

[7] Z. M. Ali, M. Othman, M. R. M. Said *et al.*, "Two Fast Computation Algorithms for Luc Cryptosystems," in Proceedings of the International Conference on Electrical Engineering and Informatics Institut Teknologi Bandung, Indonesia, 2007, pp. 434-437.

[8] Z. M. Ali, M. Othman, M. R. M. Said *et al.*, "An Efficient Computation Technique for Cryptosystems Based on Lucas Functions," in Computer and Communication Engineering, 2008. ICCCE 2008. International Conference on, 2008, pp. 187-190.

[9] Z. M. Ali, M. Othman, M. R. M. Said *et al.*, "Computation of Cryptosystem Based on Lucas Functions Using Addition Chain," in Information Technology (ITSim), 2010 International Symposium in, 2010, pp. 1082-1086.

[10] V. I. Arnold, and A. Avez, *Ergodic Problems in Classical Mechanics*, New York: Benjamin, 1968.

[11] J. C. Bajard, L. S. Didier, and P. Kornerup, "An Rns Montgomery Modular Multiplication Algorithm," *IEEE Transactions on Computers,* vol. 47, no. 7, pp. 766-776, 1998.

[12] H. Berghel, and L. O'Gorman, "Protecting Ownership Rights through Digital Watermarking," *IEEE Comput. Mag.,* vol. 29, no. 7, pp. 101–103, 1996.

[13] D. J. Bernstein, "Proving Tight Security for Standard Rabin-Williams Signatures," http://cr.yp.to/sigs/rwtight-20030926.ps, [06/05/2011, 2003].

[14] A. Bosselaers, R. Govaerts, and J. Vandewalle, "Comparison of Three Modular Reduction Functions," in Advances in Cryptology-CRYPTO'93, LNCS 773, 1993, pp. 175-186.

[15] C. Burnikel, and J. Ziegler, *Fast Recursive Division*, MPI Informatic research report, 1998.

[16] C. Burnikel, and J. Ziegler, *Fast Recursive Division*, Max-Planck-Institut fuer Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, 1998.

[17] E. Çelebi, M. Gözütok, and L. Ertaul, "Implementations of Montgomery Multiplication Algorithms in Machine Languages," in Proceedings of the 2008 International Conference on Security & Management, Las Vegas, Nevada, USA, 2008, pp. 491-497.

[18] S. Y. Chiou, and C. S. Laih, "An Efficient Algorithm for Computing the Luc Chain," in Computers and Digital Techniques, IEE Proceedings 1995, pp. 263-265.

[19] S. Y. Chiou, and C. S. Laih, "An Efficient Algorithm for Computing the Luc Chain," *Computers and Digital Techniques, IEE Proceedings -,* vol. 147, no. 4, pp. 263-265, 2000.

[20] G. C. T. Chow, K. Eguro, W. Luk *et al.*, "A Karatsuba-Based Montgomery Multiplier," in 2010 International Conference on Field Programmable Logic and Applications (FPL), 2010, pp. 434-437.

[21] S. A. Cook, "On the Minimum Computation Time of Functions.," Doctoral Thesis, Harvard University, 1966.

[22] J. T. Cross, "The Euler's Φ-Function in the Gaussian Integers," *Amer. Math.,* vol. 55, pp. 518-528, 1983.

[23] I. Damgård, S. Dussé, and B. Kaliski, "A Cryptographic Library for the Motorola Dsp56000," *Advances in Cryptology — Eurocrypt '90*, Lecture Notes in Computer Science, pp. 230-244: Springer Berlin / Heidelberg, 2006.

[24] Z. Dawei, C. Guanrong, and L. Wenbo, "A Chaos-Based Robust Wavelet-Domain Watermarking Algorithm," *Chaos, Solitons and Fractals,* vol. 22, no. 1, pp. 47-54, 2004.

[25] R. Dedekind, *Theory of Algebraic Integers*: Cambridge University Press 1996.

[26] W. Diffie, and M. Hellman, "New Directions in Cryptography," *Information Theory, IEEE Transactions on,* vol. 22, no. 6, pp. 644-654, 1976.

[27] A. El-Kassar, M. Rizk, N. Mirza *et al.*, "El-Gamal Public-Key Cryptosystem in the Domain of Gaussian Integers," *Int J Appl Math,* vol. 7, no. 4, pp. 405-412, 2001.

[28] A. N. El-Kassar, R. A. Haraty, Y. A. Awad *et al.*, "Modified Rsa in the Domains of Gaussian Integers and Polynomials over Finite Fields," in Proceedings of the ISCA 18th International Conference on Computer Applications in Industry and Engineering, Hawaii, USA, 2005, pp. 298-303.

[29] S. E. Eldridge, and C. D. Walter, "Hardware Implementation of Montgomery's Modular Multiplication Algorithm," *IEEE Transactions on Computers,* vol. 42, no. 6, pp. 693-699, 1993.

[30] H. Elkamchouchi, K. Elshenawy, and H. Shaban, "Extended Rsa Cryptosystem and Digital Signature Schemes in the Domain of Gaussian Integers," in Proceedings of the 8th International Conference on Communication Systems, 2002, pp. 91-95.

[31] M. Fürer, "Faster Integer Multiplication," in Proceedings of the thirty-ninth annual ACM symposium on Theory of computing, San Diego, California, USA 2007.

[32] P. Garrett, *Making, Breaking Codes: Introduction to Cryptography*: Prentice Hall, 2001.

[33] D. M. Gordon, "A Survey of Fast Exponentiation Methods," *Journal of Algorithms,* vol. 27, pp. 129-146, 1998.

[34] T. Granlund. "The Gnu Multiple Precision Arithmetic Library," 09/12/2010; http://gmplib.org/gmp-man-5.0.1.pdf.

[35] R. A. Haraty, A. N. El-Kassar, and H. Otrok, "A Comparative Study of Rsa Based Cryptographic Algorithms," in ISCA 13th International Conference on Intelligent and Adaptive Systems and Software Engineering, Nice, France, 2004, pp. 183-188.

[36] R. A. Haraty, A. N. El-Kassar, and H. Otrok, "A Comparative Study of El-Gamal Based Cryptographic Algorithms," *RITA,* vol. 12, no. 1, pp. 7-22, 2005.

[37] R. A. Haraty, A. N. El-Kassar, and B. Shibaro, "A Comparative Study of Rsa Based Digital Signature Algorithms," *Journal of Mathematics and Statistics* vol. 2, no. 1, pp. 354-359, 2006.

[38] R. A. Haraty, H. Otrok, and A. N. Kassar, "Attacking Elgamal Based Cryptographic Algorithms Using Pollard's Rho Algorithm," in Proceedings of the

ACS/IEEE 2005 International Conference on Computer Systems and Applications, 2005, pp. 91.

[39] A. Hariri, and A. Reyhani-Masoleh, "Bit-Serial and Bit-Parallel Montgomery Multiplication and Squaring over Gf(2^M)," *IEEE Transactions on Computers,* vol. 58, no. 10, pp. 1332-1345, 2009.

[40] K. Hasselström, "Fast Division of Large Integers," Department of Numerical Analysis and Computer Science, Royal Institute of Technology, Stockholm, Sweden, 2003.

[41] H.-C. Huang, Y.-H. Chen, and A. Abraham, "Optimized Watermarking Using Swarm-Based Bacterial Foraging," *Journal of Information Hiding and Multimedia Signal Processing,* vol. 1, no. 1, pp. 51-58, Jan. 2010 2010.

[42] M. Joye, and J. J. Quisquater, "Efficient Computation of Full Lucas Sequences," *Electronics Letters,* vol. 32, no. 6, pp. 537–538, 1996.

[43] A. Karatsuba, and Y. Ofman, "Multiplication of Many-Digital Numbers by Automatic Computers," *Proceedings of the USSR Academy of Sciences,* vol. 145, pp. 293–294, 1962.

[44] D. E. Knuth, *The Art of Computer Programming*, 3rd ed.: Addison-Wesley, 1998.

[45] Ç. K. Koc, "Analysis of Sliding Window Techniques for Exponentiation," *Computers and Mathematics with Applications,* vol. 30, pp. 17-24, 1995.

[46] Ç. K. Koc, T. Acar, and B. S. K. Jr., "Analyzing and Comparing Montgomery Multiplication Algorithms," *IEEE Micro,* vol. 16, pp. 26-33, 1996.

[47] A. Koval, "On Lucas Sequences Computation," *Int'l J. of Communications, Network and System Sciences* vol. 2, no. 12, pp. 943-944 2010.

[48] A. Koval, F. Y. Shih, and B. S. Verkhovsky, "A Pseudo-Random Pixel Rearrangement Algorithm Based on Gaussian Integers for Image Watermarking," *Journal of Information Hiding and Multimedia Signal Processing,* vol. 2, no. 1, pp. 60-70, 2010.

[49] A. Koval, and B. Verkhovsky, "Analysis of Rsa over Gaussian Integers Algorithm," in Fifth International Conference on Information Technology: New Generations (ITNG 2008), Las Vegas, Nevada, USA, 2008, pp. 101-105.

[50] A. Koval, and B. S. Verkhovsky, "On Discrete Logarithm Problem for Gaussian Integers," in International Conference on Information Security and Privacy (ISP-09), Orlando, Florida, USA, 2009, pp. 79-84.

[51] C.-S. Laih, F.-K. Tu, and W.-C. Tai, "On the Security of the Lucas Function," *Inf. Process. Lett.,* vol. 53, no. 5, pp. 243-247, 1995.

[52] A. K. Lenstra, Daniel Bleichenbacher, and W. Bosma, “Some Remarks on Lucas-Based Cryptosystems ” in 15th Annual International Cryptology Conference Santa Barbara, California, USA, 1995, pp. 386-396.

[53] C.-C. Lin, and P.-F. Shiu, “Highcapacity Data Hiding Scheme for Dct-Based Images,” *Journal of Information Hiding and Multimedia Signal Processing,* vol. 1, no. 3, pp. 220-240, July 2010, 2010.

[54] A. J. Menezes, P. C. v. Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*: CRC Press, 1997.

[55] P. L. Montgomery, “Modular Multiplication without Trial Division,” *Mathematics of Computation,* vol. 144, no. 170, pp. 519–521, 1985.

[56] M. Othman, E. M. Abulhirat, Z. M. Ali *et al.*, “A New Computation Algorithm for a Cryptosystem Based on Lucas Functions,” *Journal of Computer Science,* vol. 4, no. 12, pp. 1056-1060, 2008.

[57] M. Othman, E. M. Abulkhirat, M. R. M. Said *et al.*, “An Improvement of Luc2 Cryptosystem Algorithm Using Doubling with Remainder,” in Computing & Informatics, 2006. ICOCI '06. International Conference on, 2006, pp. 1-4.

[58] M. O. Rabin, *Digitalized Signatures and Public Key Functions as Intractable as Factorisation*, Massachusetts Institute of Technology, 1979.

[59] R. L. Rivest, A. Shamir, and L. Adleman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems,” *Commun. ACM,* vol. 21, no. 2, pp. 120-126, 1978.

[60] A. Schönhage, and V. Strassen, “Multiplikation Großer Zahlen,” *Computing*, no. 7, pp. 281–292, 1971.

[61] F. Y. Shih, *Digital Watermarking and Steganography: Fundamentals and Techniques*, Boca Raton, FL, USA: Taylor & Francis Group, CRC Press, Inc., 2008.

[62] P. Smith, “Luc Public Key Encryption: A Secure Alternative to Rsa,” *Dr. Dobb's J.,* vol. 18, no. 1, pp. 44-49, 1993.

[63] P. Smith, “Cryptography without Exponentiation,” *Dr. Dobb's J.*, no. 4, pp. 26-30, April 01, 1994, 1994.

[64] B. Verkhovsky, “Generalized Baby-Step Giant Step Algorithm for Discrete Logarithm Problem,” *Advances in Decision Technology and Intelligent Information Systems,* vol. IX, no. IIAS, pp. 88-89, 2008.

[65] B. Verkhovsky, and A. Koval, “Cryptosystem Based on Extraction of Square Roots of Complex Integers,” in Fifth International Conference on Information

Technology: New Generations (ITNG 2008), Las Vegas, Nevada, USA, 2008, pp. 1190-1191.

[66] B. Verkhovsky, and A. Mutovic, "Primality Testing Algorithm Using Pythagorean Integers," *Computer Science and Information System*, pp. 143-157, June, 2005.

[67] B. Verkhovsky, and K. Sauraj, "Quaternion-Based Primality Testing Algorithm," in Proc. Int'l Computer Science and Infor. Systems Conf., Athens,Greece, 2005.

[68] C.-T. Wang, C.-C. Chang, and C.-H. Lin, "A Method for Computing Lucas Sequences," *Computers & Mathematics with Applications,* vol. 38, no. 11-12, pp. 187-196, 1999.

[69] Y. Wu, and F. Y. Shih, "Digital Watermarking Based on Chaotic Map and Reference Register," *Pattern Recognition,* vol. 40, no. 12, pp. 3753-3763, Dec. 2007, 2007.

[70] K. Yamamoto, and M. Iwakiri, "Real-Time Audio Watermarking Based on Characteristics of Pcm in Digital Instrument," *Journal of Information Hiding and Multimedia Signal Processing,* vol. 1, no. 2, pp. 59-71, Apr. 2010, 2010.

[71] Y.-J. Yan, W.-W. Zhu, E.-P. Duan *et al.*, "A Novel Fault Resistant Algorithm for Montgomery Multiplication," in 2010 10th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT), 2010, pp. 2025-2027.

[72] Z. Yantao, M. Yunfei, and L. Zhiquan, "A Robust Chaos-Based Dct-Domain Watermarking Algorithm," in Proceedings of the 2008 International Conference on Computer Science and Software Engineering, 2008, pp. 935 - 938.

[73] G. Ye, "Image Scrambling Encryption Algorithm of Pixel Bit Based on Chaos Map," *Pattern Recognition Letters,* vol. 31, no. 5, pp. 347-354, 2010.

[74] S. M. Yen, and C. S. Laih, "Fast Algorithms for Luc Digital Signature Computation," *IEE Proceedings Computers and Digital Techniques,* vol. 142, no. 2, pp. 165-169, 1995.